PLANIFICATION DES PRÉSENTATIONS MULTIMÉDIAS

par

Kaddour Boukerche

mémoire présenté au Département de mathématiques
et d'informatique en vue de l'obtention du grade de maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES
UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, juin 1998

Le 23 juin 1998 , le jury suivant a accepté ce mémoire dans sa version finale.
date

Président-rapporteur:	M.Shengrui Wang Département de mathématiques et d'informatique	_____
Membre:	M. Kacem Zéroual Département de mathématiques et d'informatique	_____
Membre:	M. Froduald Kabanza Département de mathématiques et d'informatique	_____
Membre:	M. Michel Barbeau Département de mathématiques et d'informatique	_____

Sommaire

Dû à la croissance de la complexité et du volume de l'information à communiquer, les systèmes classiques de présentation de l'information, où il faut prévoir toutes les situations à l'avance, sont devenus inadaptés. De nouvelles techniques de traitement de l'information sont nécessaires. Dans ce mémoire, nous présentons un système de présentation multimédia (PPM) basé sur la technique de la planification. Chaque présentation multimédia est composée de séquences de film (animation multimédia). Chaque séquence de film est à son tour composée d'actions de base qui sont pré-encodées dans le système. La combinaison de ces actions de base est réalisée d'une manière dynamique par un planificateur appelé *TLPLan*. Notre système procède, par la suite, à une extension de ce plan en un film. Cette transformation du plan en un film est réalisée par un système auteur appelé *Director*. L'idée de coupler le système auteur (*Director*) avec le planificateur (*TLPLan*) permet à notre système d'être adaptable et flexible. Il est adaptable parce qu'il suffit de changer la librairie des actions de base et les médias impliqués pour le faire fonctionner avec une nouvelle application. Il est flexible, car toutes les situations ne sont pas prévues à l'avance : l'usager peut formuler de nouvelles tâches à présenter et le planificateur trouve des plans qui satisfont ces tâches. Autrement dit, l'usager décrit les caractéristiques du film et non le film lui-même; c'est le planificateur qui trouve le film et PPM décide quels médias utiliser et comment les coordonner. Il est certainement plus agréable de suivre des présentations produites par notre système que de lire des pages de texte ou de dessins avec des références croisées.

Remerciements

Il m'est très agréable maintenant de pouvoir exprimer par quelques mots toute la reconnaissance que je porte à mon directeur, monsieur Froduald Kabanza, pour son appui constant et son entière participation à la réalisation de ce projet. Pour sa disponibilité, ses conseils et sa patience, je le remercie très sincèrement.

Je tiens à remercier monsieur Michel Barbeau d'avoir accepté de me co-diriger et monsieur Kacem Zéroual qui n'a cessé de stimuler mon engouement et de me prodiguer des conseils.

Il me paraît opportun de dire un grand merci à tous mes amis. J'ai été heureux de partager ces moments avec eux.

Enfin, qu'il me soit permis d'exprimer ma gratitude à mes parents, à mes frères et à mes soeurs.

Table des matières

Liste des figures

Introduction

La communication multimédia est essentielle dans notre vie quotidienne. Nous nous exprimons à l'aide d'un certain nombre de *médias*, principalement le langage parlé, les gestes et les dessins. D'autre part, notre système sensoriel reçoit des données à travers la vision, l'ouïe et le touché. Certains modes de communication peuvent être plus efficaces et mieux appropriés que d'autres pour communiquer un certain type d'information. Par exemple, une image peut être plus eloquante qu'un millier de mots, alors qu'un ensemble de mots bien choisis peut être plus descriptif qu'une image. D'autre part, le langage parlé convient mieux pour exprimer un sentiment alors qu'une carte transmet mieux l'information géographique sur un terrain.

Les présentations multimédias sont des programmes qui intègrent des types de médias variés, tels que le son, la vidéo, le texte et le graphique. Ces présentations sont conçues pour être exécutées sur des ordinateurs, des kiosques publics ou des écrans de télévision. Elles peuvent être des logiciels d'apprentissage, de la documentation en ligne, des services support clients, des jeux ou de la publicité. Ainsi par exemple, il est plus simple et plus rapide de comprendre comment faire fonctionner ou réparer un appareil (e.g., modem, moteur d'une voiture) en suivant une présentation multimédia plutôt que de lire des dizaines de pages dans un manuel. Un autre exemple assez fréquent de la vie quotidienne est l'assemblage des meubles. Ce genre de tâche est plus facile avec un guide multimédia

interactif que les utilisateurs peuvent employer dans différents modes d'interaction selon leur niveau d'agilité. Cela fait penser aux présentations télévisées (e.g., les émissions consacrées au bricolage), sauf que dans un guide multimédia il y a de l'interaction. Le fait que le guide soit programmé ouvre aussi la voie à des présentations "intelligentes", comme nous le verrons plus bas.

L'industrie des jeux électroniques et les kiosques sont d'autres exemples. Dans ces deux cas, l'usager se sent complètement impliqué dans la présentation grâce au niveau d'interactivité très élevé qu'ils offrent. Il n'est pas obligé de subir la présentation du début à la fin comme dans une présentation télévisée. Au contraire, à n'importe quel moment, il peut demander la présentation d'une autre tâche, ce qui veut dire que l'ordre de déroulement de la présentation n'est pas linéaire : il y a des branchements qui permettent aux utilisateurs de s'aiguiller en fonction de leurs choix.

Il y a quelques années, développer des applications multimédias s'avérait une tâche coûteuse et particulièrement difficile, réservée aux spécialistes. Heureusement, la progression rapide des technologies multimédias a réduit le coût et augmenté la capacité du traitement des supports multimédias, matériels et logiciels. Par exemple, les cartes vidéo, de son et graphiques sont de plus en plus performantes et de moins au moins coûteuses. D'autre part, on dispose de logiciels dotés d'outils d'aide et d'assistance qui facilitent le développement des applications multimédias, tels que le système *Director*. De tels systèmes sont appelés *systèmes auteurs*.

Les systèmes auteurs permettent de développer des systèmes de présentation multimédia dans tous les domaines où le besoin de communiquer l'information d'une manière efficace se manifeste. Ainsi, plusieurs entreprises commerciales ou industrielles, de même

que des institutions éducationnelles proposent aux visiteurs des kiosques de présentations multimédias, pour se faire connaître et promouvoir leurs produits et services. Ces kiosques sont faciles à développer à l'aide des systèmes auteurs.

Cependant, la plupart des systèmes multimédias actuels sont statiques dans la mesure où toutes les situations y sont pré-encodées. Si l'utilisateur fait un choix qui n'est pas prévu dans le système, sa requête est tout simplement ignorée. Pour les domaines d'application où le nombre d'interactions désirées n'est pas très élevé, ces systèmes sont appropriés. Hélas, il y a beaucoup d'applications où cette hypothèse n'est pas vérifiée.

Dans ce travail, nous présentons un système de planification des présentations multimédias (PPM) qui génère automatiquement et en temps réel des présentations multimédias. Réaliser cette tâche revient à répondre à deux questions : quoi présenter et comment le présenter. Le système PPM a non seulement la capacité de déterminer les séquences qui composent la présentation, mais plus encore, ces séquences sont générées en temps réel dès que le besoin d'expliquer une tâche est formulé. Chaque séquence est composée d'actions de base qui sont pré-encodées dans le système. Par contre, leur combinaison est réalisée d'une manière dynamique par un programme appelé *planificateur*; la séquence est considérée comme un plan d'actions en prévision des événements qui seront générés par l'utilisateur qui interagit avec le système.

Reprenons l'exemple de l'assemblage d'une bibliothèque à partir d'actions de base très simples telles que : placer un morceau de meuble dans une certaine position, raccorder un morceau de meuble avec un autre. Le planificateur peut générer des combinaisons d'actions dont l'exécution permet d'assembler le meuble ou des parties du meuble selon les exigences individuelles des usagers. Autrement dit, le planificateur trouve dans l'espace des combinaisons possibles des actions, celles qui rencontrent les besoins des utilisateurs.

Après que le planificateur ait livré un plan d'actions, le système procède à la coordination des médias impliqués et à la présentation des instructions d'assemblage.

Notre travail comprend, d'une part, la réalisation d'une interface graphique conviviale qui constituée d'un ensemble de menus qui permettent à l'usager de :

- choisir parmi plusieurs applications disponibles ou en ajouter de nouvelles,
- exprimer des besoins pour lesquels le système doit fournir des explications multimédias,
- maintenir l'ensemble des tables de correspondance entre les actions de base qui composeront la présentation et les médias impliqués, ainsi que les tables qui contiennent l'historique des présentations déjà réalisées (configurations de départ, configurations finales et plans).

D'autre part, nous avons réalisé des processus chargés de transmettre les tâches à présenter à un planificateur et de récupérer un plan d'actions à partir duquel une présentation multimédia cohérente est générée.

Notre méthodologie pour atteindre ces objectifs a été de coupler un système de présentation multimédia classique (*Director*) avec un planificateur (*TLPLan*). Il est important de souligner que le planificateur est complètement indépendant du système de présentation multimédia. Cela permet à notre système d'être facilement adaptable : pour l'adapter à une application différente, il suffit de changer la librairie des actions de base et les médias impliqués. Cependant, l'idée de coupler un planificateur avec un système de présentation multimédia n'est pas nouvelle en soi. Par exemple, des systèmes similaires et plus complexes ont été réalisés par André et Rist [3, 5, 9] . Notre système est originale parce qu'il utilise un système auteur différent (*Director*) et un planificateur efficace (*TLPLan*). Ainsi, notre système constitue une base pour un logiciel offrant plus de possibilités.

En résumé, notre contribution est le développement d'un logiciel qui intègre *Director* et *TLPLan*. Ce système est générique. Il permet de remplacer facilement TLPLan par un ou plusieurs autres planificateurs. On peut aussi écrire des librairies pour intégrer et manipuler tous les objets médias afin de remplacer *Director*.

Dans ce mémoire, nous commençons par décrire *Director* et *TLPLan*. Ensuite, nous exposons l'architecture de notre système en décrivant ses principales composantes, leurs fonctionnalités essentielles et leurs interactions. Ensuite, nous présentons des exemples ayant servi d'expérimentation de notre système : la présentation d'un appareil photo et la présentation de la navigation d'un personnage animé dans un ensemble de chambres contiguës. Nous concluons par les leçons apprises de ce projet et les extensions futures.

Chapitre 1

Le système auteur Director

Dans ce chapitre, nous décrivons *Director* avant de voir, dans des chapitres ultérieurs, comment il est combiné avec un système de planification pour obtenir un outil de présentation multimédia automatisé tel qu'expliqué précédemment dans l'introduction.

Dans la communication personne à personne, le succès de la présentation dépend de la rhétorique et des capacités didacticielles du présentateur. Le présentateur doit percevoir la satisfaction ou l'insatisfaction de l'audience et adopter les comportements nécessaires pour la satisfaire. Généralement, ceci peut se faire par l'intermédiaire de l'interaction du présentateur avec l'audience (*ex. :* questions et réponses). Ce principe d'interaction est aussi cruciale dans la communication *personne-machine*, sinon davantage. Dans les présentations multimédias, l'interactivité permet à l'usager de visualiser uniquement les séquences de film qui l'intéressent. Donc, l'interactivité personnalise les présentations. Par exemple, dans une présentation d'un cours d'algèbre, si nous ne nous intéressons qu'à l'inversion d'une matrice, il est souhaitable que la présentation permette de visualiser uniquement ce thème et non toutes les notions d'algèbre contenues dans cette animation. Cette manière de tailler les présentations aux exigences des utilisateurs permet à ces derniers de se concentrer uniquement sur les sujets qui les intéressent. Généralement l'usager

fait ses sélections par l'intermédiaire de menus organisés selon un arbre de choix. Cette possibilité d'offrir plusieurs choix à l'usager est appelée la navigation. Ainsi, contrairement à une présentation linéaire qui impose à l'audience un ordre pré-défini, la navigation permet des choix arborescents, des sauts d'un sujet à un autre et des retours en arrière.

De nos jours, plusieurs langages de programmation permettent de manipuler plusieurs médias (*ex. :* JAVA). Ils permettent aussi de réaliser des animations sophistiquées. Néanmoins, pour jongler avec les librairies de ces langages de programmation, des programmes relativement complexes doivent êtres codés. Par contre, les systèmes auteurs spécialisés dans le développement des applications multimédias offrent un niveau de programmation plus élevé, de façon qu'on puisse même écrire une animation interactive sans avoir à écrire la moindre ligne de code. Bien sûr, là aussi on se sert éventuellement des langages de programmation, mais seulement pour réaliser des tâches encore plus complexes.

Pour la réalisation de notre outil, nous avons choisi *Director* [17]. *Director* est le système auteur le plus utilisé pour le développement des applications multimédias interactives, pour *Windows* et *Machintosh*. Les applications multimédias créées sous *Director* sont appelées *films*. La composition d'un film sous *Director* nécessite d'abord l'intégration des objets médias impliqués. Pour intégrer les médias, il faut les importer et les stocker dans une structure appelée la table des casts (*acteurs*). Ensuite, ces médias sont disposés dans des trames qui vont constituer les différentes scènes du film que *Director* anime. La table qui contient ces trames est appelée la table de montage. Contrairement à un film classique (*ex. :* film de télévision) ces trames peuvent être animées dans un ordre non séquentiel. Le langage de programmation *Lingo* gère les sauts d'une trame à une autre. Il contrôle les objets sur la scène, les faire apparaître, dispparaître ou changer de place.

Dans la figure 1 on remarque aussi deux autres composantes : la scène d'animation

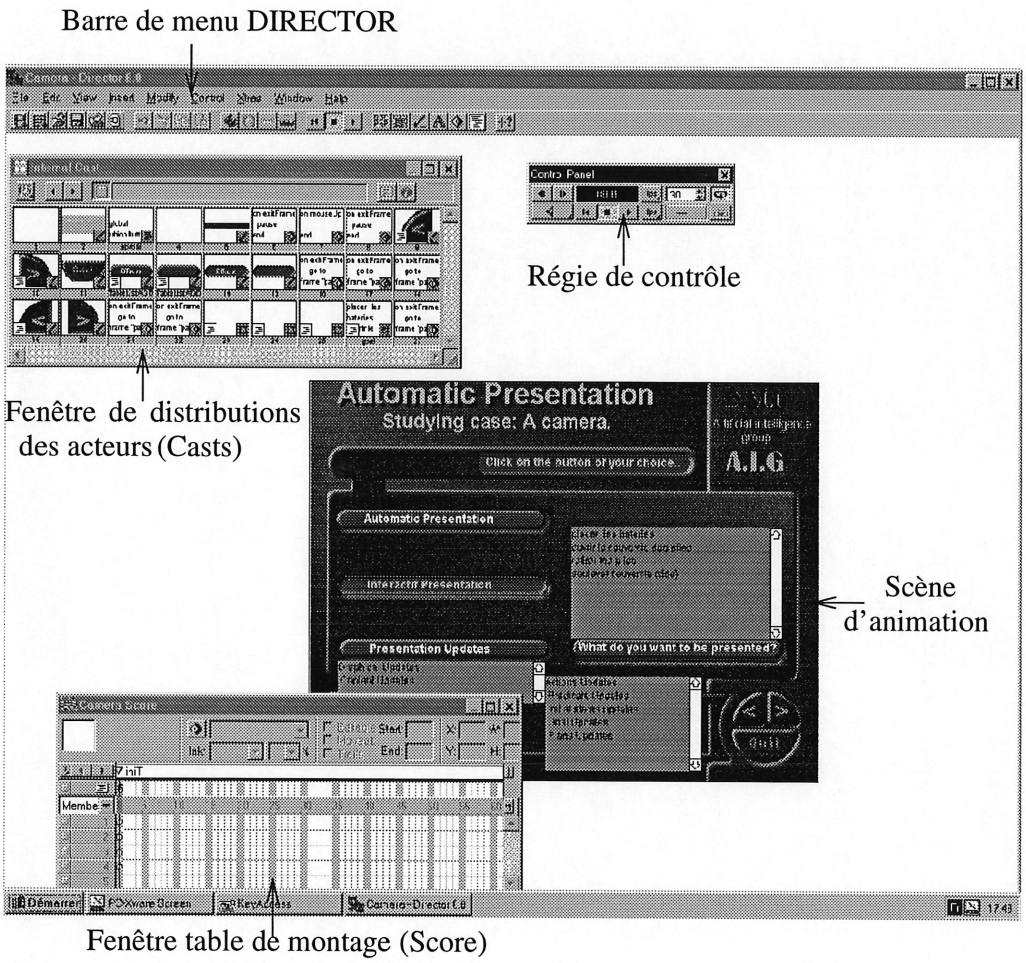

FIG. 1 – *Fenêtre du système Director*

et la régie de contrôle. La scène est la fenêtre dans laquelle les animations de *Director* apparaîssent. Les dimensions de la scène sont spécifiées par l'usager. La régie est similaire aux boutons d'un magnétoscope; elle permet d'avancer, reculer ou arrêter une animation. Elle peut aussi contrôler le son, la cadence d'exécution des trames et boucler sur une animation (voir figure 2). Par exemple, il suffit de cliquer sur la touche *lecture* pour voir jouer sur l'écran une animation multimédia qui explique comment faire pour retirer les piles d'un appareil photo.

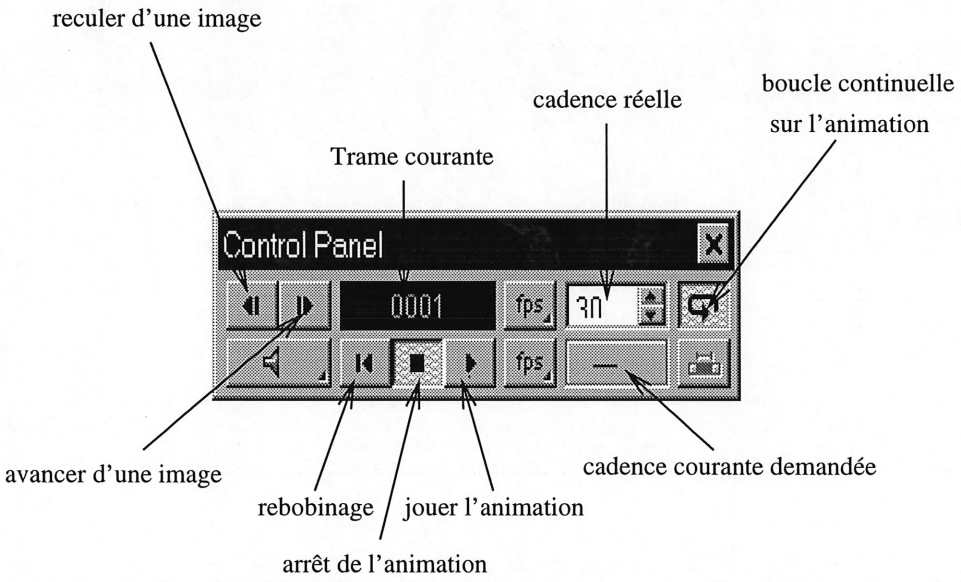

FIG. 2 – *Fenêtre de la régie du système Director*

Pour mieux comprendre ce que *Director* permet de faire, il est plus facile de commencer par décrire plus en détails ses trois composantes principales : la table des casts, la table de montage et le langage de programmation *Lingo*.

1.1 La table des casts

Un cast est tout simplement un élément entrant dans la composition d'un film. Il peut être un graphique, une séquence vidéo, un texte, un enregistrement son ou musical, une palette de couleurs, un bouton ou un programme *Lingo*. La table des casts est une structure où tous les casts sont stockées. Cette table est construite de deux façons :

- On peut créer les casts en utilisant des outils propres à *Director* : outils de dessin, outils d'enregistrement de sons ou d'édition de programmes.
- On peut aussi importer des casts créés par d'autres applications comme *PhotoShop, Adobe Première ou SoundEdit.*

Director permet la visualisation de la table des casts telle qu'illustrée dans la figure 3. Les casts 25 et 32 contiennent des séquences de vidéo et de son nécessaires pour composer un film expliquant comment ouvrir le couvercle de la chambre des piles d'un appareil photo. Les casts 26 et 33 sont nécessaires pour composer un film expliquant comment retirer les piles de leur chambre. Les casts 27 et 34 sont nécessaires pour composer un film expliquant comment refermer le couvercle de la chambre des piles. Avec tous les médias contenus dans les casts, on peut monter un film général expliquant toutes les fonctions d'un appareil photo. Ce film sera constitué de séquences de films contenues dans des casts, organisées selon un programme plus ou moins complexe programmé dans le langage *Lingo*. Par exemple, en fonction des choix de sujets entrés par l'usager (*ex. :* "comment changer les piles d'un appareil photo"), le programme décidera quels casts activer pour composer ce film. Pour monter le film, nous avons besoin des éléments de base (*c-à-d.,* les casts) mais aussi d'un outil pour les synchroniser (table de montage) et écrire des programmes pour les coordonner (programmes *Lingo*). Dans ce qui suit, nous considérons d'abord la table de montage.

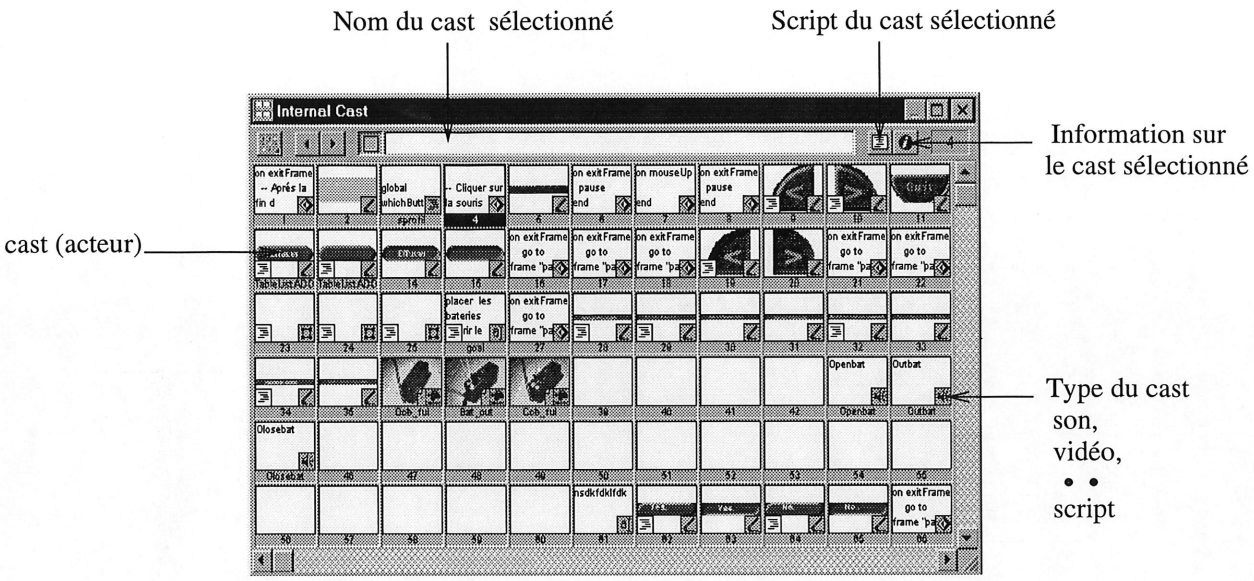

FIG. 3 – *Table de distribution des casts (acteurs).*

1.2 La table de montage

La table de montage est une matrice de cellules qui garde une trace de la position de chaque cast sur la scène. Une cellule est la plus petite unité de la table de montage. Elle représente le contenu d'un élément d'une trame à un instant donné. La table de montage nous permet de connaître le comportement de chaque cast à chaque instant de l'animation. Elle nous informe sur les casts qui sont sur la scène et sur leurs positions. Par exemple, la figure 4 montre le score pour le film de la présentation d'un appareil photo : chaque colonne décrit une trame de cellules du film à un instant donné, autrement dit c'est tout ce qu'on voit sur la scène lorsque le film atteint ce point. Un canal est une ligne de cellules. Chaque canal peut être dédié à un type d'information spécifique, pour faciliter la manipulation des casts. Par exemple, dans notre système le canal 17 est dédié au bouton qui active l'animation, ce qui facilite son identification dans les scripts *Lingo*.

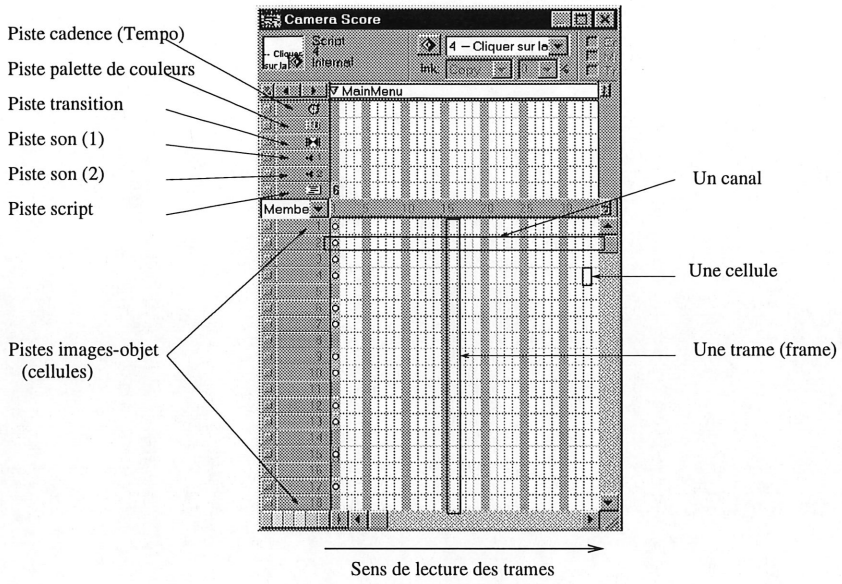

FIG. 4 – *Table de montage (score)*

Nous n'allons pas détailler toutes les composantes de la table de montage, mais seulement donner quelques notions suffisantes pour comprendre la partie essentielle de notre outil, à savoir, l'intégration de la planification avec *Director*.

Pour animer les casts qui se trouvent dans la table des casts, ont doit les ajouter dans la table de montage afin que *Director* puisse les jouer si ce sont des films, les activer si ce sont des programmes, les afficher si se sont des textes ou des graphiques. Ajouter un cast dans la table de montage revient à créer un lien entre la cellule qui va contenir ce cast et le cast proprement dit qui se trouve dans la table des casts. Donc, dans la table de montage, on trouve les images des casts et non les casts eux mêmes ce sont ces images, appelées *sprites*, que *Director* anime sur la scène. Comme nous l'avons déjà mentionné, c'est la trame qui indique la position temporelle du cast. Ainsi, si un cast précède un autre dans le score, alors le premier cast apparaîtra avant le second. On met un cast dans une cellule particulière en le glissant dedans par la souris ou indirectement par des

instructions d'un programme *Lingo*.

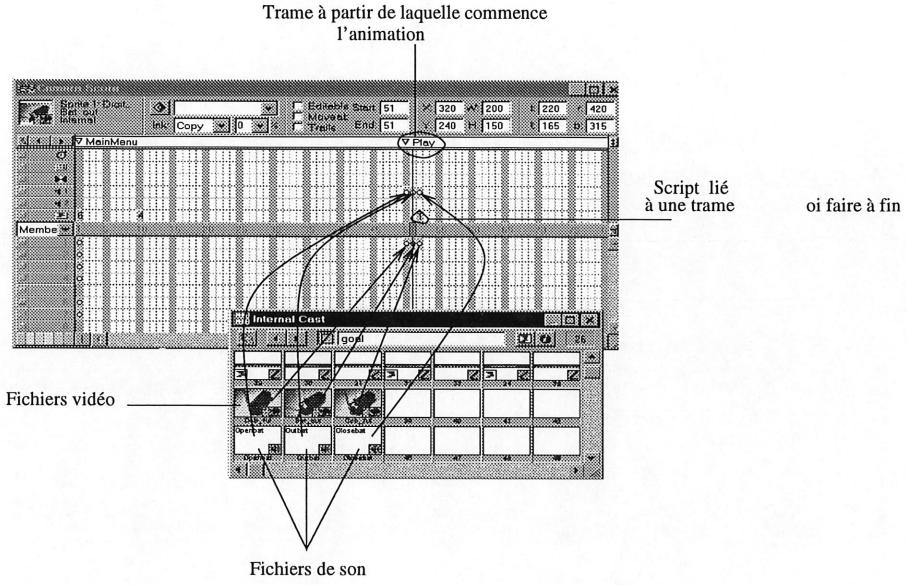

FIG. 5 – *Ajout des casts dans la table de montage*

Dans ce travail, nous appelons un objet, tout élément nécessaire pour la composition d'un film. Par conséquent, tout cast est un objet. Les programmes *Lingo* peuvent contrôler un film *Director* en spécifiant la vitesse de la lecture des trames, le choix des palettes de couleurs pour les différentes scènes du film et le mode de transition d'une trame à une autre. Par l'intermédiaire de menus, *Director* permet de spécifier ces contrôles de manière complètement interactive dans les pistes correspondantes. La figure 4 montre les différentes pistes disponibles dans la table de montage, les six premières pistes permettent de spécifier les contrôles cités ci-dessus et les 48 autres permettent d'inclure les objets qui constituent le film et les programmes qui déterminent leurs comportements.

Dans la suite du texte, nous allons résumer les fonctionnalités principales de chaque piste. La piste de cadence permet de contrôler la vitesse de cadence de lecture des trames,

de placer des pauses dans un film et de préciser comment les annuler pour que le film puisse continuer à s'exécuter (*ex. :* par un clique d'une souris ou par la frappe d'une touche du clavier) et d'exprimer des contraintes telles que : attendre l'exécution du fichier son avant de passer à la lecture de la trame suivante. La piste de palettes de couleurs permet de choisir les palettes de couleurs pour chaque trame. La piste de transitions permet de définir le mode de transition d'une scène à une autre. Par exemple, le glissement de la scène courante du haut vers le bas et en même temps l'apparition de la nouvelle scène. Les deux pistes de sons contiennent les sons qu'on veut utiliser dans l'application multimédia. La piste de scripts associée à chaque trame permet de gérer les événements qui se produisent au niveau de cette trame. Les pistes des images-objet contiennent les casts impliqués dans l'animation. *Director* a toute une panoplie d'effets spéciaux pour enjoliver les présentations.

Après avoir expliqué comment intégrer les médias dans *Director*, il est nécessaire d'expliquer comment animer ces médias et comment en faire des films. Pour cela, il faut définir la notion de scripts et leur rôle dans un film. Pour approfondir ces deux aspects, nous abordons dans la section suivante le langage de programmation *Lingo*.

1.3 Le langage de programmation Lingo

Le langage de programmation *Lingo* permet d'écrire des programmes (ou scripts) qui coordonnent les casts et leur présentation dans le temps par :

- la prise en compte des actions des utilisateurs (interactivité),
- la manipulation de texte,
- le contrôle de son,
- l'importation de données externes dans une animation,

- l'exécutions d'animations indépendantes ou interactives,
- l'évaluation d'équations mathématiques,
- la communication avec des objets externes à *Director* et
- la communication avec des éléments d'interface tels que les boutons et les menus.

Lingo est un langage orienté/objet basé sur la transmission de messages. Les messages peuvent être internes (générés par les programmes *Lingo*) ou externes (générés par une interface utilisateur ou une interface d'une autre application). Les scripts varient dans leur complexité. Ils peuvent consister en une instruction à un seul mot ou un ensemble d'instructions plus complexes. On appelle *handler* un srcipt attaché à un objet. Les handlers permettent aux objets d'interagir avec d'autres objets ou des usagers. Ils sont écrits par le développeur. Chaque handler commence par le mot clé *On* suivi par le nom qui l'identifie, les instructions, et se termine par *End*. Ils existe aussi des handlers pré-définis dans *Lingo* tel que *ExitFrame* qui est un script associé à une trame et qui est activé à la fin de son exécution. Dans les scripts *Lingo* les lignes de commentaires sont précédées par un double tiret "––". Les trames de la table de montage peuvent être identifiées par des libellés ou leurs numéros de séquence dans la table de montage. Ceci permet à *Lingo* d'effectuer des sauts d'une trame à une autre. Par exemple, le script de la figure 6 spécifie que lorsque le film atteint la fin de la trame à laquelle ce script est associé, l'animation doit faire un saut à une autre trame étiquettée *"MainMenu"*.

```
On ExitFrame
        -- À la sortie de cette trame l'animation revient
        -- au menu principal pour effectuer d'autres choix.
        go to frame "MainMenu"
End
```

FIG. 6 – *Premier exemple de script*

Le script illustré dans la figure 7 est associé à un cast. On y trouve l'essentiel des instructions d'un langage comme *JAVA*. Dans cet exemple on remarque deux handlers. Le premier handler, *mouseDown*, est associé à l'événement "enfoncer le bouton de la souris" qui consiste à enfoncer un bouton sur la scène. Dès que cet événement est déclenché, le script correspondant change la couleur du bouton sur la scène pour montrer qu'il est activé. Le deuxième handler, *MouseUp*, est associé à l'événement "relacher le bouton de la souris". Le script associé à cet événement restaure la couleur initiale du bouton sur la scène et fait appel au handler *processClick* qui détermine le comportement que le système doit adopter suite à cet événement.

```
On MouseDown
        -- Rendre le sprite du bouton de l'activation de l'animation actif
        set the puppet of sprite 15 to TRUE

        -- Rendre visible le bouton de l'activation de l'animation
        set the castNum of sprite 20 to 21

        -- Porter ces modifications sur la scène
        updateStage

        -- Souris appuyée n'a aucun effet.
        repeat while the mouseDown
        end repeat
End

On mouseUp
        -- Rendre visible le bouton de l'activation de l'animation
        -- enfoncé
        set the castNum of sprite 15 to 10
        updateStage

        -- Passer l'événement clique de la souris au handler
        -- spécialiste du traitement des cliques
        processClick
End
```

FIG. 7 – *Deuxième exemple de script*

Lingo utilise quatre types de scripts : script de *cast*, de *trame*, de *sprite* et de *film*. Quand un événement se produit, un message est envoyé aux handlers primaires (gestionnaires primaires d'événements) dédiés à traiter cet événement et le script associé est exécuté. Par la suite, le message est propagé à d'autres objets jusqu'à ce qu'une instruction arrête la propagation du message. Si une telle instruction n'est pas rencontrée ou les handlers primaires n'existent pas, alors le message est propagé dans l'ordre suivant : au sprite, au cast, à la trame et enfin au film. À chaque niveau, on peut arrêter la propagation du message aux autres objets. La figure 8 illustre la hiérarchie de propagation d'un message suite à un événement.

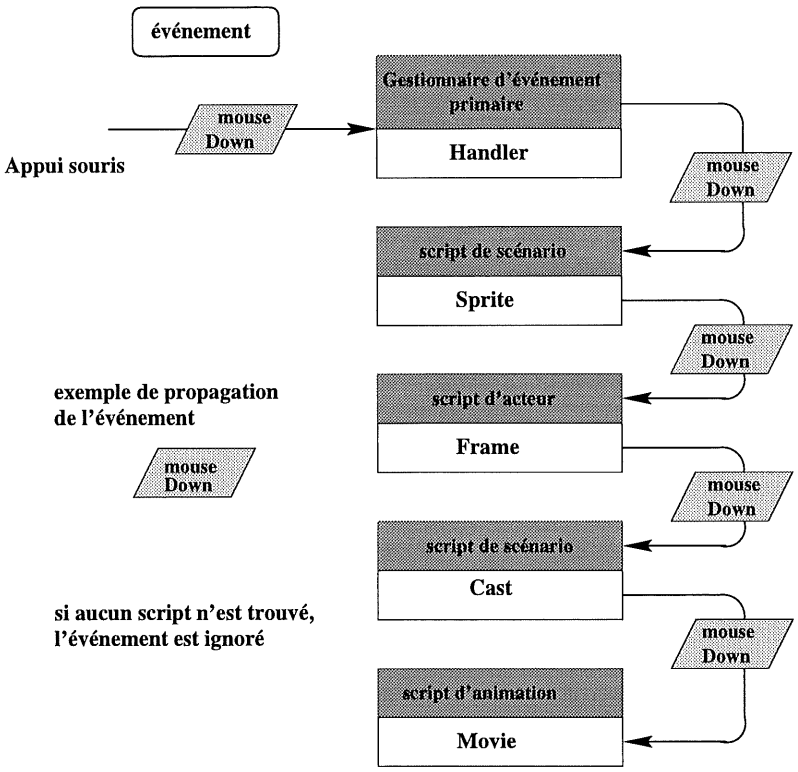

FIG. 8 – *La hiérarchie de propagation d'un message dans Director*

Il arrive parfois que l'on atteigne les limites du langage *Lingo*. Par exemple, si on veut utiliser les services d'une base de données multimédias, il faut se rabattre sur un langage de programmation externe à *Lingo* et rédiger des procédures dans ce langage (*ex. :* C++ ou C). *Lingo* permet de faire appel à des applications externes écrites dans d'autres langages. En particulier, nous nous en servons pour faire appel aux services du planificateur *TLPLan* et aux services du système de gestion des bases de données multimédias *V12* [22]. Le système *V12* contient tous les médias qui sont impliqués dans la réalisation des présentations. Comme dans tout système de gestion de base de données, *V12* est doté d'un langage de requête pour réaliser des opérations classiques telles que créer une base de données, ajouter, détruire ou modifier un enregistrement.

En résumé, *Director* est un logiciel qui offre plusieurs outils pour programmer des applications multimédias, tels que des kiosques ou des systèmes de présentation. Nous l'avons choisi pour plusieurs raisons :

- sa simplicité : il n'est pas nécessaire d'être un spécialiste pour pouvoir développer une application multimédia simple;
- sa capacité d'intégrer et de manipuler différents types de médias;
- son ouverture aux autre systèmes : il est facile de l'interfacer avec des applications externes écrites en d'autres langages.

Maintenant que nous avons expliqué *Director* et avant de décrire PPM, il convient d'expliquer d'abord ce que c'est un planificateur. Pour être concret, nous parlerons du planificateur *TLPLan* qui est à la base de ce travail.

Chapitre 2

Le planificateur TLPLan

Un planificateur est essentiellement un programme qui génère d'autres programmes, appelés plans, pour accomplir une tâche donnée. Le mot "plan" suggère que ces programmes sont obtenus en analysant des prédictions (ou simulations) de l'interaction future du système avec son environnement. Quelque soient le système et la tâche considérés, un planificateur génère un plan à partir de données d'entrée composées d'actions (ou instructions) de base exécutables dans le système à un certain niveau d'abstraction, une description de la tâche et une configuration initiale du système. Une tâche consiste à amener le système à travers des configurations désirables, c'est-à-dire, des configurations satisfaisant une propriété appelée but. Un plan est alors obtenu comme une séquence d'actions de base qui génèrent une séquence de configurations satisfaisant le but.

Puisque nous nous intéressons uniquement aux présentations multimédias, les actions de base sont des scripts *Lingo*. Le niveau d'abstraction correspond à la complexité des scripts. Ainsi, on peut donner un but décrivant les caractéristiques d'un film désiré et le planificateur va nous donner un script qui combine les scripts de base pour réaliser l'animation de ce film. Ce qui est intéressant ici c'est que l'on décrit les caractéristiques du film et non le film lui-même. C'est au planificateur de construire le film. En d'autres

termes, on décrit ce qu'on veut comme film sans trop savoir comment le programmer, c'est-à-dire quels casts utiliser et comment les coordonner. Ceci confère un caractère "intelligent" à la présentation. Nous avons utilisé un planificateur appelé *TLPLan*, développé par Bacchus et Kabanza [11].

Comme la plupart des planificateurs, le logiciel *TLPLan* prend comme arguments un état initial, une description des schémas d'actions de base et un but. Il retourne un plan qui est une séquence d'actions transformant l'état initial en un état satisfaisant le but.

Considérons l'exemple de l'appareil photo et supposons qu'on veuille savoir quelles sont les actions à exécuter pour retirer les piles de leur chambre. Au départ, le couvercle de la chambre des piles est fermé; cette situation est considérée comme l'état initial. Pour réaliser cette tâche, le planificateur renvoie le plan suivant : ouvrir le couvercle de la chambre des piles, soulever le couvercle de la chambre des piles et retirer les piles de leur chambre.

Remarquons que chaque action ne s'applique que si certaines conditions sont vérifiées et, qu'une fois l'action exécutée, l'état courant de l'appareil photo est changé. Par exemple, lorsqu'on applique l'action "ouvrir le couvercle de la chambre des piles", l'état de l'appareil change : le couvercle de la chambre des piles qui était fermé devient ouvert. Ceci rend les conditions de l'application de l'action "soulever le couvercle de la chambre des piles" vraies et à son tour l'application de cette action rend les conditions de l'application de l'action"retirer les piles de leur chambre" vraies. Lorsque cette dernière action est exécutée, la chambre des piles devient vide, ce qui satisfait le but désiré.

Intuitivement, un état est une description de l'environnement à un instant donné. Dans notre système, cette description est contenue dans *Director*; plus précisément, elle

est décrite par la trame courante et les valeurs des variables d'états des scripts *Lingo*. Par exemple, le couvercle de la chambre des piles peut être dans l'état fermé ou ouvert.

En résumé, une action correspond à un film dont l'exécution est capable de changer l'une ou l'autre des composantes de la situation courante. En d'autres termes, une action représente une transition de la situation actuelle à une nouvelle situation. Finalement, un plan est un ensemble de séquences de films dont l'exécution explique comment réaliser la tâche désirée.

Il est connu que tout programme écrit dans n'importe quel langage peut être modélisé par un système de transition entre états où un état décrit les valeurs des variables et le compteur d'instructions [16]. Dans ce contexte une exécution particulière correspond à un chemin, c'est-à-dire une séquence de transitions ou, en d'autres mots, un plan.

La vision derrière un planificateur est un peu analogue. On voit le système qu'on veut contrôler comme un système discret qui opère en faisant une action à la fois dans son état courant. Évidemment, une fois l'action effectuée, on change d'état et dans le nouvel état on effectue une autre action. Ainsi de suite, jusqu'à ce qu'on atteigne l'état désirable. Le problème c'est qu'à chaque état, on peut appliquer plusieurs actions différentes. Par exemple, dans le contexte des systèmes multimédias, si la trame actuelle affiche une scène donnée, l'état suivant sera décrit par la programmation d'une nouvelle trame où les casts correspondants à l'action sont ajoutés pour constituer la nouvelle scène. Selon l'action que l'on choisit, on obtient un déroulement différent du film, c'est-à-dire une séquence d'états différents. En analysant la séquence d'états, on peut déterminer si oui ou non elle satisfait le but. Par exemple, si le but est de décrire comment ouvrir le couvercle d'un appareil photo, une séquence d'états serait satisfaisante si elle se termine par un état où l'image affichée est celle d'un appareil photo avec son couvercle ouvert.

Afin de décrire mécaniquement cette propriété, on s'arrange pour décrire les actions en leur associant des propriétés pertinentes. En fait, tout planificateur utilise un langage de description des actions permettant de raisonner sur les propriétés associées aux actions. Dans *TLPLan*, chaque action est décrite par trois composantes : précondition, suppression et addition [10]. La précondition est une liste de conditions qui doivent être vraies pour que l'action soit exécutable. La suppression et l'addition sont aussi des listes de conditions, sauf qu'elles décrivent les effets de l'action, c'est-à-dire quel est l'état résultant.

La transformation de l'état courant n'est possible que par la vérification de certaines conditions. Après l'exécution d'une action, certaines conditions qui étaient vraies dans l'état précédent deviennent fausses dans l'état courant et il faut les supprimer de la description de l'état courant. Cependant, de nouvelles conditions viennent décrire cet état courant et il faut les ajouter dans sa description. C'est pour ça que les effets sont décrits par une liste de suppressions et une liste des ajouts.

Par exemple, pour pouvoir placer les piles dans leur chambre il faut les préconditions suivantes : le couvercle de la chambre des piles doit être soulevé et la chambre des piles doit être vide. Une fois que les piles sont placées dans leur chambre, la condition "chambre des piles vide" devient fausse et donc il faut la supprimer de la description de l'état actuel. En plus, "la chambre des piles n'est pas vide" est une nouvelle condition qui devient vraie, et donc, il faut l'ajouter dans la description de l'état actuel. Notons que si on applique une action à un état où les préconditions sont fausses, il n'en résulte aucune transformation.

TLPLan trouve un plan en essayant d'analyser toutes les combinaisons possibles des

actions. Ceci se traduit par la recherche dans l'espace d'états possibles. Pour ce faire, *TLPLan* a besoin en entrée d'une liste de toutes les actions possibles. Si *TLPLan* ne trouve pas, cela ne veut pas dire qu'il n'existe pas de plan satisfaisant le but, mais tout simplement que les actions fournies à *TLPLan* ne permettent pas de composer un tel plan. C'est pourquoi il est important de donner à *TLPLan* une liste d'actions aussi exhaustive que possible. Évidemment, plus d'actions on a, plus l'espace d'états possibles est grand et plus le calcul du plan risque de prendre du temps. Ce phénomène est connu sous le nom de l'explosion combinatoire. *TLPLan* possède un mécanisme original pour limiter l'explosion combinatoire. Nous reviendrons sur cet aspect à la fin de ce chapitre.

Afin de permettre des descriptions concises des actions, *TLPLan* accepte aussi des schémas d'actions paramétrables et pas simplement des actions complètement définies. Dans un schéma d'une action, les paramètres sont remplacés par des variables qui seront affectées par des valeurs spécifiques au moment de la recherche du plan. Ces variables n'ont rien à avoir avec les variables des programmes *Lingo*. Par exemple, on peut définir l'action (`ouvrir` x) où x peut être remplacé au moment de la planification par le couvercle de la chambre des piles, le couvercle de la chambre du film ou le cache de l'objectif, dépendamment de la tâche à planifier.

Il est important de remarquer que finalement les états générés sont des états incluant uniquement des conditions ou des propriétés décrites dans les actions. Les actions sont des modèles abstraits des vrais programmes *Lingo*. Par conséquent, les actions sont des modèles abstraits des états décrits dans *Director*. Le lien entre les actions de *TLPLan* et les programmes *Lingo* sera détaillé dans le chapitre 3.

Après avoir expliqué comment *TLPLan* génère les séquences d'états, on va voir comment il s'en sert pour trouver un plan satisfaisant le but. Tout d'abord, un but est expliqué

comme une formule de logique vérifiable sur des séquences d'états. Il n'est pas nécessaire d'entrer dans les détails de définition de cette logique pour comprendre la suite de ce mémoire. Notons simplement qu'étant donné un but, *TLPLan* a un mécanisme pour vérifier, au fur et à mesure qu'il génère des séquences d'états, si ces séquences sont conformes au but. Dès qu'une séquence satisfaisant le but est trouvée, *TLPLan* arrête d'explorer le reste de l'espace d'états.

Considérons une partie de la modélisation de la présentation de l'appareil photo, c'est-à-dire comment faire pour placer les piles dans leur chambre. Une modélisation complète sera détaillé dans le chapitre 4 . En premier, nous énumérons les conditions (ou prédicats) qui constituent les listes de précondition, de suppression et d'addition :

1) (Ouvert ?x), x est le couvercle de l'appareil photo,
2) (Fermé ?x),
3) (Baissé ?x),
4) (Soulevé ?x) et
5) (ChambrepilesVide).

Pour modéliser ce problème, nous avons considéré trois actions : ouvrir le couvercle de la chambre des piles, soulever le couvercle de la chambre des piles et placer les piles dans leur chambre. La figure 9 explicite le schémas de chaque action en spécifiant les préconditions, les suppressions et les additions.

Dans cet exemple, nous considérons que dans la situation initiale le couvercle de la chambre des piles est fermé et les piles sont logées dans leurs chambre (*c-à'd.*, la chambre des piles n'est pas vide). Ceci se traduit dans le langage de *TLPLan* par :

(`Fermé` CouvercleChambrepiles) $\wedge \neg$ (ChambrepilesVide)

Puisque nous voulons retirer les piles de leur chambre (ceci signifie rendre la chambre des piles vide), dans le langage de *TLPLan* ce but se traduit par :

(ChambrepilesVide)

`Ouvrir le couvercle de la chambre des piles`
1) Nom action : (Ouvrir ?x)
2) Préconditions : (Fermé ?x)
3) Additions: (Ouvert ?x)
4) Suppressions : (Fermé ?x)

`Soulever le couvercle de la chambre des piles`
1) Nom action : (Soulever ?x)
2) Préconditions : (Baissé ?x) $\wedge$ (Ouvert ?x)
3) Additions : (Soulevé ?x)
4) Suppressions : (Baissé ?x)

`Retirer les piles de leur chambre`
1) Nom action : (Retirer ?x)
2) Préconditions : $\neg$(ChambrepilesVide)
3) Additions : (ChambrepilesVide)
4) Suppressions : $\neg$(ChambrepilesVide)

FIG. 9 – *La liste des schémas d'actions*

Ayant toutes ces données en entrée, *TLPLan* calcule le plan suivant :

1) (`Ouvrir` CouvercleChambreBatterie)
2) (`Soulever` CouvercleChambreBatterie)
3) (`Retirer` piles)

Une fois qu'un plan d'actions de ce genre est généré, on peut l'étendre avec des médias pour en faire un film.

Afin de démontrer la généralité de l'idée de la planification des applications multimédias, il convient de considérer une autre application : la navigation d'un personnage de dessin animé que nous avons appelé amicalement *Toto*, dans un ensemble de chambres contiguës qui communiquent entre elles par des portes. Le but de cette application est de permettre à *Toto* de se déplacer d'une chambre à une autre.

Dans cet exemple, nous avons considéré trois actions : passage de *Toto* d'une chambre à une autre chambre contiguë, ouvrir une porte et fermer une porte.

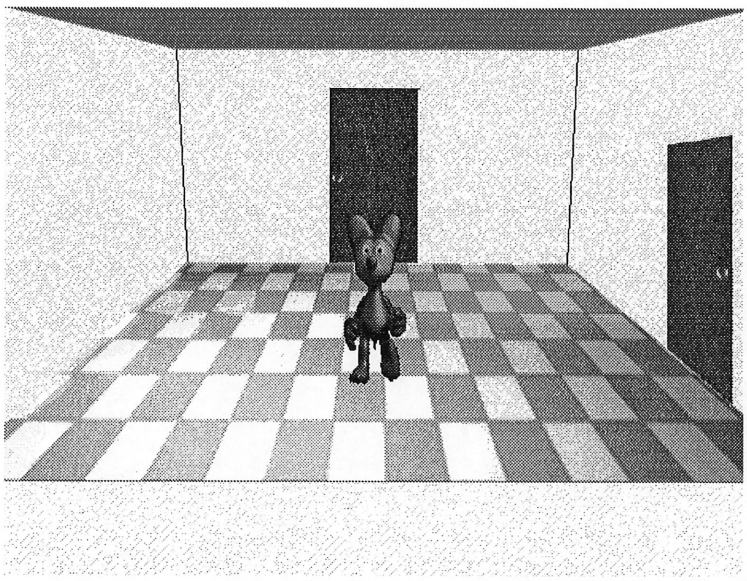

FIG. 10 – *Le personnage animé Toto dans la chambre 1*

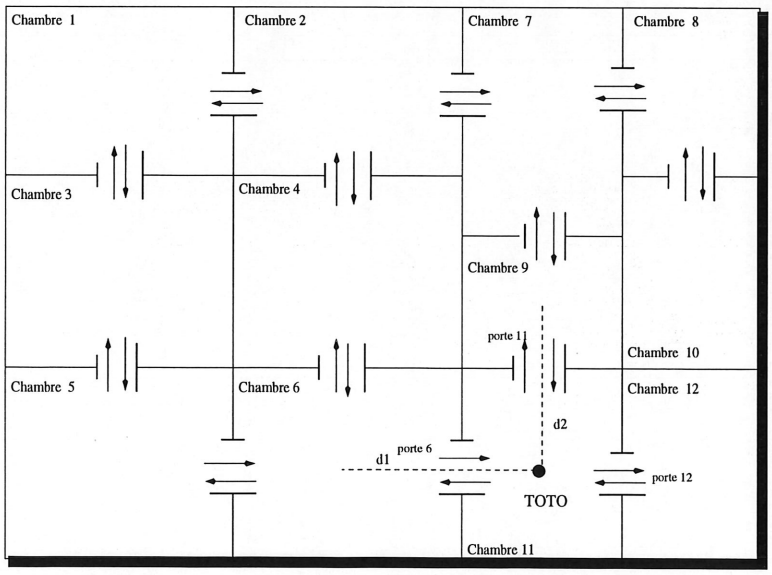

FIG. 11 – *L'environnement du personnage animé Toto.*

Dans la figure 11, *Toto* est dans la chambre 11, mais dès que *Toto* se déplace dans une autre chambre connexe, c'est cette nouvelle chambre qui est affichée. La figure 11 montre un plan de toutes les chambres.

Supposons qu'on veuille déplacer *Toto* dans la chambre 9. Trois actions sont possibles : aller dans la chambre 6, aller dans la chambre 9 ou aller dans la chambre 12. *TLPLan* dispose d'un mécanisme original pour guider la recherche. Ce mécanisme utilise les stratégies de contrôle de recherche exprimées en logique temporelle qui permet à *Toto*, dans cet exemple, d'aller directement dans la chambre 9 et d'éviter les chambres 6 et 12. Donc, éviter de perdre du temps dans une recherche inutile.

Cet exemple démontre les aptitudes de *TLPLan* à planifier efficacement des présentations multimédias. Lorsque nous avons privé *TLPLan* de son mécanisme qui le guide dans la recherche et nous avons fait évoluer *Toto* dans un ensemble de 20 chambres et plus. Nous avons constaté que les performances de la planification des tâches se dégradent d'une manière considérable (*c-à-d.*, le planificateur est de 5 fois à 10 fois moins rapide). Dans l'article [11], il est clairement démontré comment ce mécanisme accélère la recherche.

Maintenant que nous avons des idées précises sur *Director* et *TLPLan*, nous allons expliquer dans le chapitre suivant comment ces deux logiciels sont couplés dans notre système de planification des présentations afin de générer automatiquement et en temps réel des présentations multimédias.

Chapitre 3

Architecture du système PPM

Dans les deux chapitres précédents, nous avons présenté les supports logiciels nécessaires pour le développement de notre système. Dans ce chapitre, nous parlerons de l'architecture de PPM et des scénarios de sa mise en œuvre. Pour faciliter la compréhension de toutes les étapes du processus de génération des présentations multimédias, nous allons aborder le problème en deux parties. Dans la première partie, nous allons présenter une vue fonctionnelle de PPM dans le but de délimiter ses fonctions et de déterminer les acteurs avec lesquels il interagit. Dans la deuxième partie, nous allons présenter l'architecture de PPM et détailler toutes ses composantes. À la fin de ce chapitre, PPM ne devrait avoir aucun secret pour le lecteur.

3.1 Vue fonctionnelle du système PPM

Comme nous l'avons expliqué, la tâche de PPM est de générer une présentation multimédia à partir d'une tâche spécifiée par un usager, d'un état initial, d'une liste d'actions, d'une base de données multimédias et des contraintes de présentation.

L'exemple de la figure 12 montre les paramètres en entrée pour le système PPM ainsi

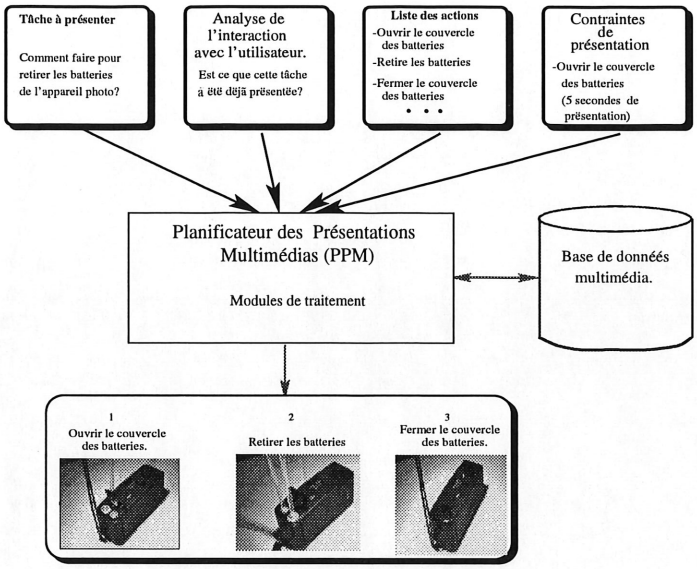

FIG. 12 – *Vue fonctionnelle du système PPM*

que la sortie qu'il génère. La sortie générée est un film *Director* qui explique à l'usager toutes les manœuvres nécessaires pour pouvoir retirer les piles d'un appareil photo de leur chambre. Les tâches à présenter sont proposées à l'usager dans un menu. Une fois une tâche sélectionnée, PPM vérifie si cette tâche est formulée pour la première fois ou, au contraire, si c'est une tâche qui a déjà été expliquée. Dans les deux cas, une présentation multimédia est livrée à l'intention de l'usager. Cependant, dans le cas où la tâche a été présentée plusieurs fois auparavant, PPM conclu que cette présentation n'est convaincante. Il produit de nouvelles explications multimédias, dans l'espoir qu'elles satisfassent d'avantage l'usager. Cette propriété de vérification des tâches est désignée dans PPM par l'analyse des interactions PPM-usager.

Pour composer la présentation multimédia, PPM utilise les services de *TLPLan*, qui, en combinant les actions de base, trouve la séquence d'actions qui satisfait le but. À partir de cette séquence d'actions, PPM extrait de la base de données multimédias les médias

correspondants à chaque action. Ensuite, il procède à la fusion et à la coordination entre ces médias en respectant des contraintes. Ces contraintes portent sur le temps maximal d'animation alloué à chaque action. Ainsi, PPM peut être perçu comme un processus de transformation d'une demande d'un usager en un film *Director*.

3.2 Architecture du système PPM

Pour mettre en œuvre le processus de génération des présentations multimédias, nous avons conçu une architecture qui capte toutes les fonctionnalités du système. Pour obtenir une telle architecture, nous avons du, tout d'abord, identifier ses fonctions. Lorsque nous considérons la vue globale et externe de PPM nous savons qu'il interagit avec l'usager et *TLPLan*. L'examen des interactions de PPM avec ces deux entités détermine les fonctions qu'il doit réaliser.

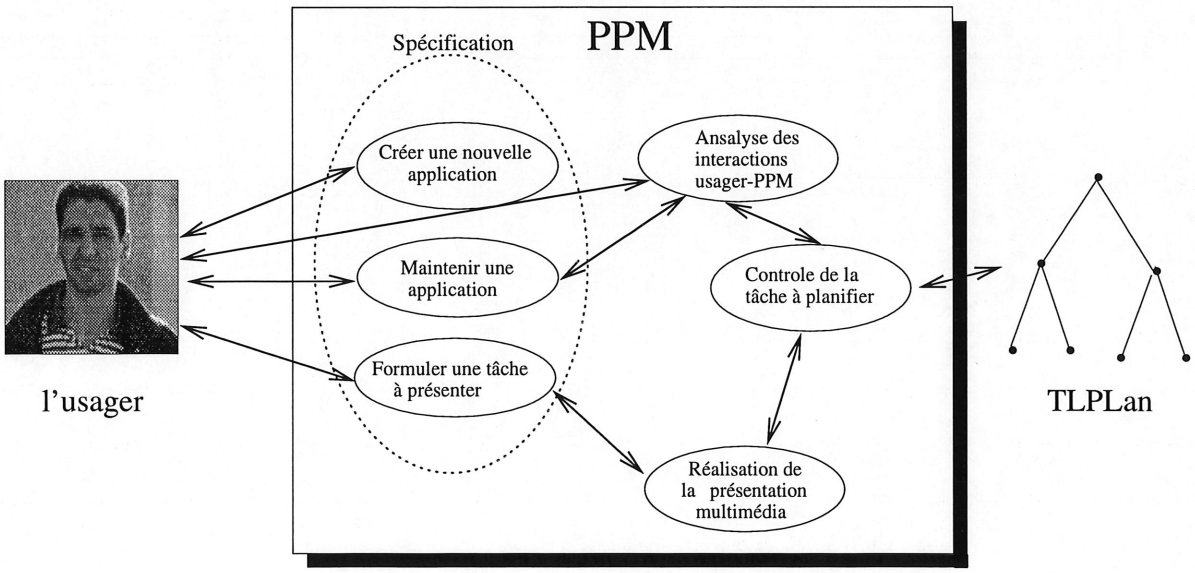

FIG. 13 – *Interaction PPM-usager et PPM-TLPLan*

Dans la figure 13, nous remarquons que PPM permet à l'usager de créer, de maintenir des applications et de formuler des tâches à présenter. Ces trois fonctionnalités constituent l'essentiel des interactions usager-PPM. La composante *Spécification* représente l'interface qui va permettre ces interactions. Le but ultime de PPM est de générer des présentations qui satisfont l'usager. C'est pourquoi nous l'avons doté de la composante *Analyse* afin de déceler les présentations non satisfaisantes et de les remplacer par des nouvelles. Après avoir choisi et analysé la tâche à présenter, l'étape suivante consiste à trouver un plan pour cette tâche. L'interaction PPM-*TLPLan* est assurée par la composante *Contrôle.* La composante *Contrôle* communique la tâche à planifier à *TLPlan* et récupère le plan calculé par ce dernier. Le film réel de *Director* est composé sur la base des modèles abstraits, qui sont les séquences d'actions fournies par *TLPLan.* La composante *Réalisation* transforme les médias associés aux actions du plan en une présentation multimédia cohérente. Le processus de transformation est un processus de sélection, de fusion et de coordination entre les médias. L'étude des interactions PPM-*TLPLan* et PPM-usager nous permet de partager le fonctionnement de PPM sur quatre composantes : *Spécification, Analyse, Contrôle et Réalisation.*

La figure 14 présente l'architecture de PPM, dont nous connaissons maintenant le fonctionnement. Cependant, nous tenons à préciser que PPM répond à un stimulus initié par l'usager. Par sa réaction, PPM agit sur son environnement, d'abord sur *Director* par la génération du film et ensuite sur l'usager en améliorant ses connaissances concernant certaines tâches. Il nous reste à détailler chaque composante, à donner un aperçu de sa mise en œuvre et à montrer ses interactions avec les autres composantes du système.

3.2.1 Composante Spécification

La composante *Spécification* est chargée de gérer toutes les interactions entre l'usager et PPM. Nous distinguons deux types d'interactions, celles qui permettent à l'usager

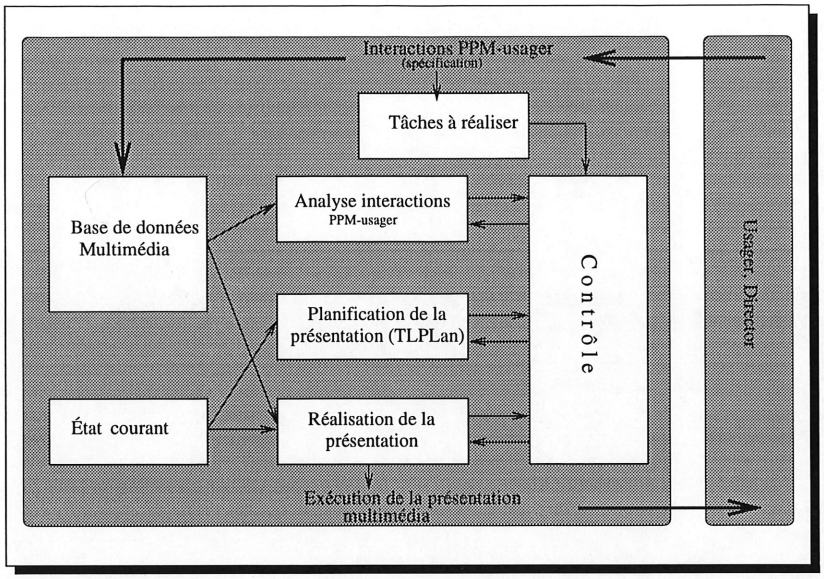

FIG. 14 – *Architecture du système PPM*

de créer et de maintenir des applications de présentation et celles qui lui permettent de sélectionner les tâches à présenter. Par conséquent, nous avons divisé cette composante en deux : la sous-composante *création et maintenance* des applications à présenter et la sous-composante *formulation* des tâches à présenter.

Sous-composante création et maintenance des applications

Le menu de la figure 15 est le premier menu qui s'affiche lors de l'exécution du système PPM. Il permet de créer des applications ou d'en choisir une déjà existante. Naturellement, la première opération à effectuer est celle de créer une application à présenter. Le menu de la figure 16 permet de spécifier le nom de l'application créée.

La création d'une nouvelle application consiste en la construction d'une base de données multimédias munie de cinq tables (voir figure 21). Un peu plus loin dans le texte, nous détaillerons la structure et le rôle de chaque table. Dès que l'usager a spécifié le

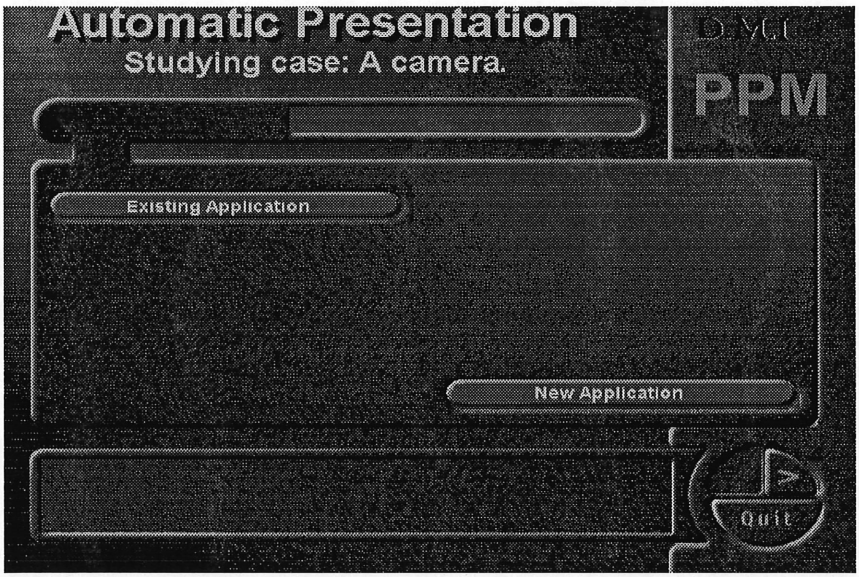

FIG. 15 – *Menu principal*

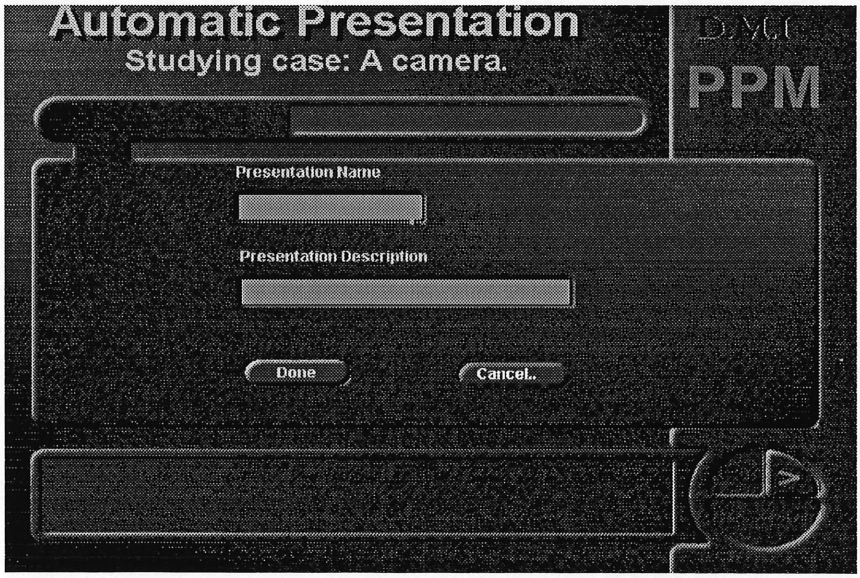

FIG. 16 – *Création d'une nouvelle application*

nom de la nouvelle application, celui-ci est aussitôt inséré dans la liste des applications disponibles dans PPM. Comme nous l'avons signalé déjà, la liste des actions de base doit être fournie à l'avance à *TLPLan*. Notre idée consiste à faire correspondre à chaque action un ensemble de médias, qui vont constituer les séquences du film final. Pour stocker ces médias, nous avons besoin d'une table *Actions*. L'interface de mise à jour de cette table est illustrée dans la figure 17.

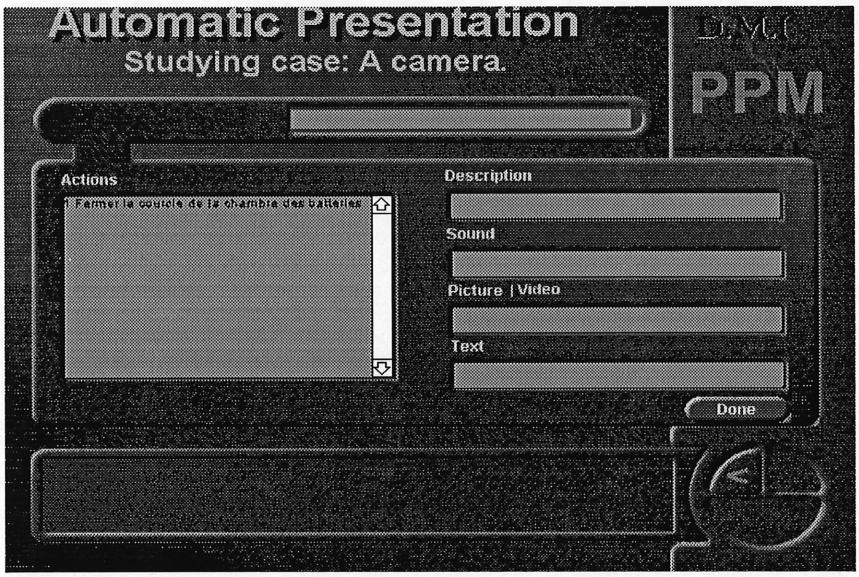

FIG. 17 – *Menu de mise à jour de la table des actions*

D'après le chapitre 2, nous savons que les prédicats servent à décrire les buts et les états initiaux. Puisque PPM doit permettre à l'usager de formuler les buts à atteindre (*c-à-d.*, les tâches à réaliser) et éventuellement les états initiaux, alors la liste des prédicats est stockée dans la table *Prédicats* conçue à cet effet. L'interface de mise à jour de cette table est illustrée dans la figure 18.

Après la création des deux tables *Actions* et *Prédicats*, la sous-composante *création*

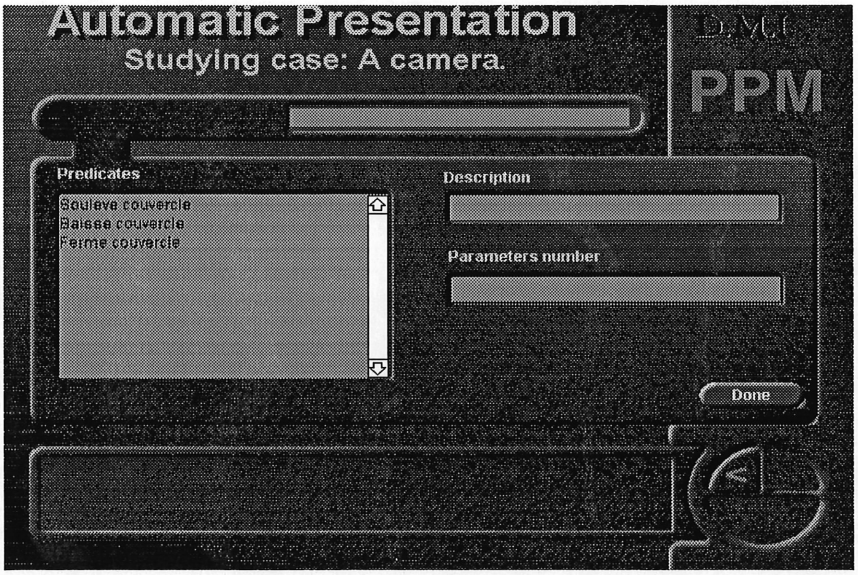

FIG. 18 – *Menu de mise à jour de la table des prédicats*

et *maintenance* des applications accède au fichier qui contient la modélisation de l'application (*i.e.,* la liste des prédicats et la liste des actions). Pour chaque action de base et chaque prédicat, le script *takeTLPLanModelisation* crée un enregistrement dans la table correspondante. Nous avons préféré extraire directement du fichier modélisation les noms des prédicats et des actions afin d'éviter les erreurs de saisie.

Nous savons que PPM offre la possibilité de formuler des états initiaux, des buts et des plans. Par conséquent, nous avons besoin de trois autres tables pour pouvoir stocker ces trois types d'information. Ces tables retiennent et mettent à jour toutes les interactions usager-PPM. D'après le chapitre 2, les tâches à présenter et les états initiaux sont décrits par des conjonctions de prédicats. La figure 19 montre l'interface de spécification d'un état-but à partir de la liste des prédicats. La spécification d'un état initial est une opération tout à fait similaire. La table de plans contient toutes les présentations déjà réalisées. Chaque enregistrement de la table *Plans* est constitué d'un

état-but, d'un état initial, d'une séquence d'actions et d'une description de cette séquence. La figure 20 montre l'interface qui permet la mise à jour d'un plan. Nous avons prévu de laisser la possibilité à l'usager de modifier lui-même un plan, dans le but de lui permettre de personnaliser ses présentations.

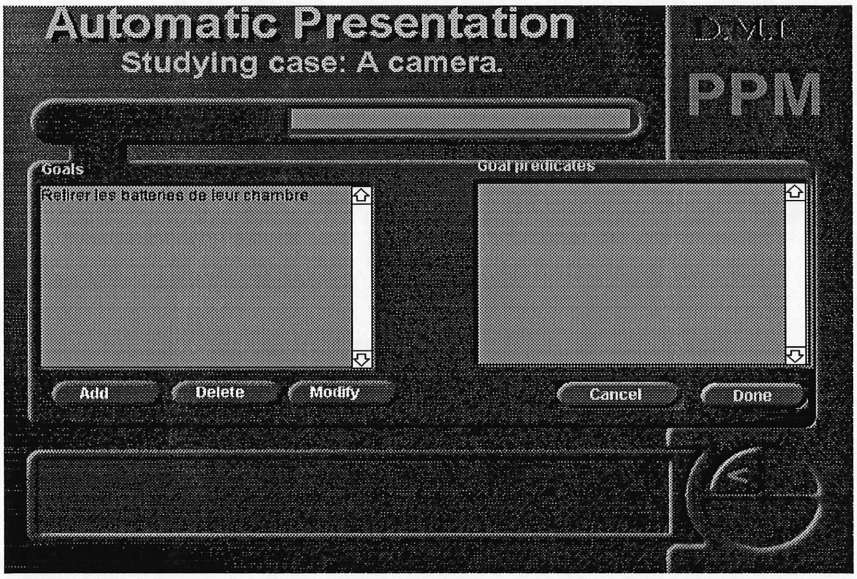

FIG. 19 – *Menu de mise à jour de la table des états-buts*

Nous savons que les scripts correspondants à chaque composante sont mis en œuvre dans le langage *Lingo*. Les programmes codés en *Lingo* sont listés dans la partie annexe de ce mémoire. Pour donner une idée bien précise de la mise en œuvre des composantes, nous allons présenter les scénarios qui sont à la base des programmes. La première fonction que doit réaliser cette sous-composante consiste à créer une base de données multimédias. Le processus responsable de la création de cette base est appelé *createDataBase*, dont

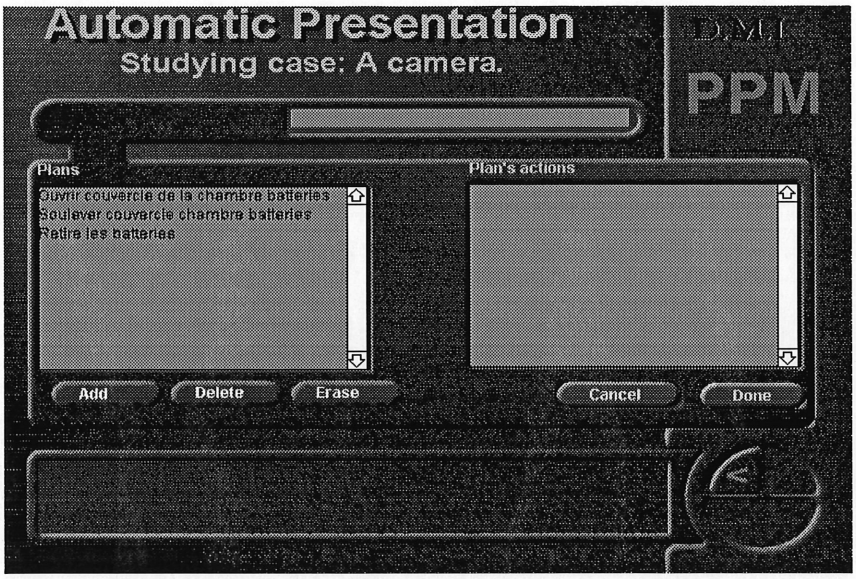

FIG. 20 – *Menu de mise à jour de la table des plans*

voici le scénario :

1) *createDataBase :* **Création d'une nouvelle application**

1.1) liste des applications ← le nom de la nouvelle application
1.2) création de la nouvelle base de données multimédias

1.2.1) création de la table `ACTIONS`
1.2.1) création de la table `PRÉDICATS`
1.2.1) création de la table `BUTS`
1.2.1) création de la table `ÉTATS-INITIAUX`
1.2.1) création de la table `PLANS`

Après la création de la base de données, la deuxième fonction de cette sous-composante consiste à remplir les tables *Actions* et *Prédicats* à partir des listes des prédicats et des

actions. Le processus responsable de la réalisation de cette tâche est appelé *getTlplan-Modelisation*, dont voici le scénario :

2) *getTlplanModelisation :* **Remplir les tables ACTIONS et PRÉDICATS**

2.1) Lecture du fichier modélisation, pour chaque action et prédicat **Répéter :**

2.1.1) table ACTIONS ← nom action
2.1.2) table PRÉDICATS ← nom prédicat, nombre de paramètres du prédicats

Le processus qui permet de formuler un but et un état initial est appelé *formulate-GoalInitialSt.* Le rôle de ce processus est de permettre à l'usager de spécifier les tâches qu'il veut visualiser et éventuellement les états de départ correspondants s'il le désire. Le scénario de ce processus est le suivant :

3) *formulateGoalInitialSt :* **Formulation d'un but ou d'un état initial**

3.1) table BUT ← liste des prédicats
3.2) table ÉTATS-INITIAUX ← liste des prédicats

Cette sous-composante permet aussi à l'usager de formuler un plan, en spécifiant les séquences d'actions. Elle lui permet aussi de modifier manuellement des plans existants. Le processus qui réalise ces tâches est appelé *formulatePlan.* Son scénario de mise en œuvre est le suivant :

4) *formulatePlan :* **Formulation d'un plan**

4.1) champ "but" de la table PLANS ← un but de la liste des buts
4.2) champ "état" initial de la table PLANS ← un état de la liste de listes états initiaux
4.3) champ "plan" de la table PLANS ← une séquence d'actions choisies de la liste des actions de base

Nous avons expliqué en quoi consiste la création d'une application sous PPM. Nous allons voir maintenant comment fonctionne la sous-composante qui permet à l'usager de choisir une tâche à présenter.

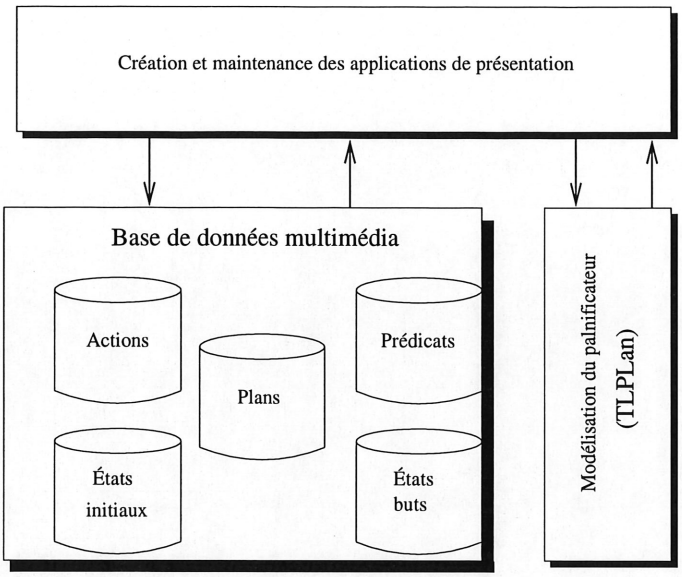

FIG. 21 – *Sous-composante de création et de maintenance des applications*

Sous-composante formulation de la tâche à présenter

Avant de choisir une tâche à présenter, l'usager doit d'abord choisir l'application qui l'intéresse. Le menu de la figure 22 permet la visualisation de la liste des applications existantes. En sélectionnant une des applications, le menu de la figure 23 s'affiche. Parmi les opérations que l'usager peut réaliser, on trouve : maintenir la base de données multimédias correspondante à l'application selectionnée, choisir une présentation automatique ou choisir une tâche à présenter. Le choix d'une présentation automatique permet à l'usager de suivre sur l'écran une seule présentation multimédia qui couvre un ensemble de tâches. Cet ensemble de tâches est déterminé à l'avance dans une liste que PPM se charge de traiter tâche après tâche. Par exemple, si l'usager choisit de voir une présentation automatique de l'application "présentation d'un appareil photo", alors un film *Director* lui est proposé pour expliquer les manœuvres nécessaires pour l'utilisation de cet appareil photo.

Choisir une tâche à présenter consiste à en sélectionner une dans la liste des tâches

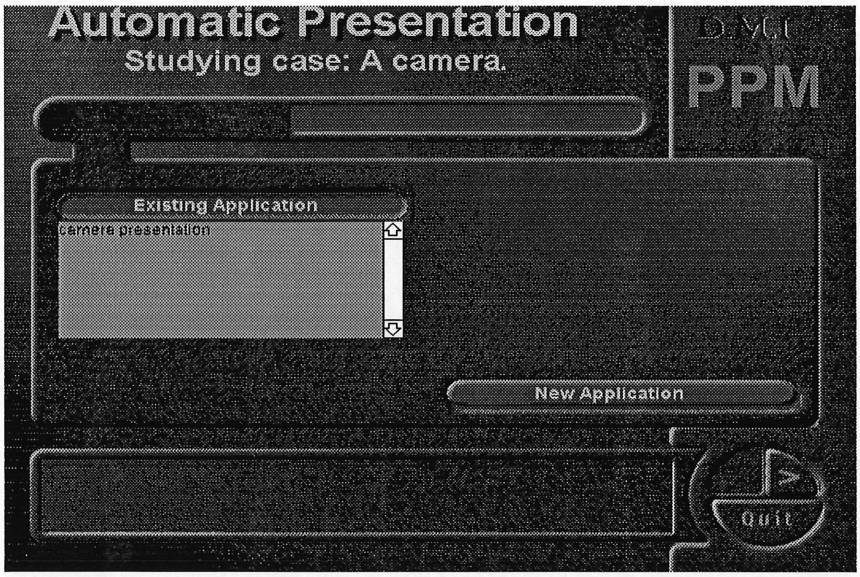

FIG. 22 – *Sélection d'une application existante*

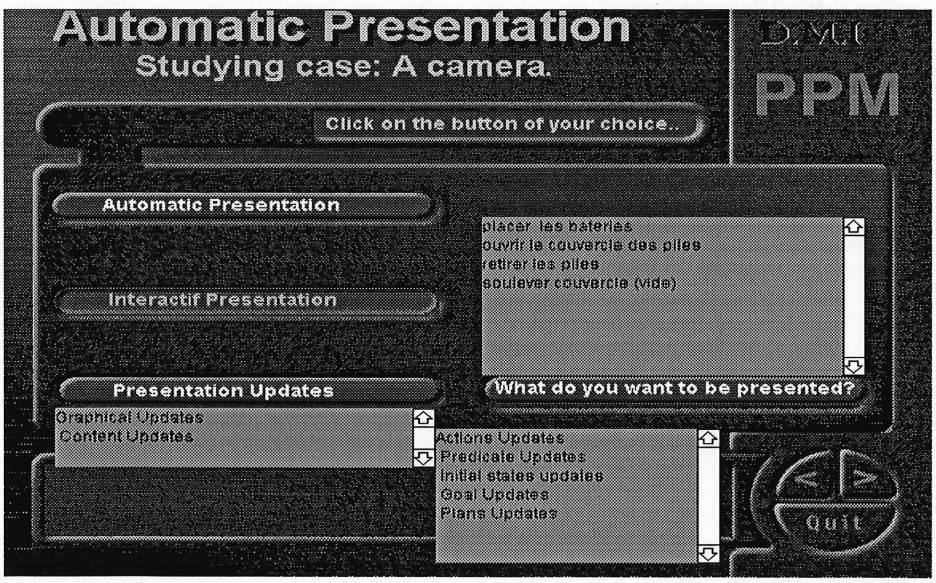

FIG. 23 – *Sélection de la tâche à présenter*

proposées. Dans l'appellation de cette sous-composante, nous avons utilisé le mot "formulation", qui semble un peu ambitieux, alors que le mot "sélection" semble plus approprié. Le mot "formulation" est choisi avec la vision future que nous projetons pour le système PPM. Cette composante fera l'objet des plus importantes extensions envisagées pour PPM. Nous en parlons dans la section "travaux futurs" de notre conclusion. La figure 23 montre la liste des tâches à présenter contenues dans la table *Buts* de la base de données multimédias. La mise en œuvre de cette composante est réalisée par le processus *"selectGoal"*, développé selon le scénario suivant :

5) *selectGoal :* **Sélection d'une tâche**

5.1) liste des tâches ← les buts se trouvant dans la table des BUTS
5.2) sélection d'une tâche

Une fois que l'usager a sélectionné une tâche, PPM procède à l'analyse de cette interaction.

3.2.2 Composante Analyse

La composante *Analyse* examine si la tâche sélectionnée a déjà été présentée durant cette même session. Dans ce cas, deux hypothèses peuvent être à l'origine de cette sélection : soit l'usager n'est pas satisfait de la présentation multimédia précédemment fournie, soit c'est une erreur (*ex. :* usager distrait). Dans le cas d'une présentation non satisfaisante, la composante *Analyse* envoie la tâche à la composante *Contrôle* avec l'option "tâche à replanifier". Autrement dit, il faut ignorer l'ancienne présentation et en planifier une nouvelle. Si la tâche est formulée pour la première fois, la composante *Analyse* communique cette tâche à la composante *Contrôle* pour être planifiée. Dans le cas où c'est une tâche formulée moins de trois fois, alors on prend le plan déjà généré dans la table des *Plans* pour monter le film correspondant. La figure 25 illustre ces différentes

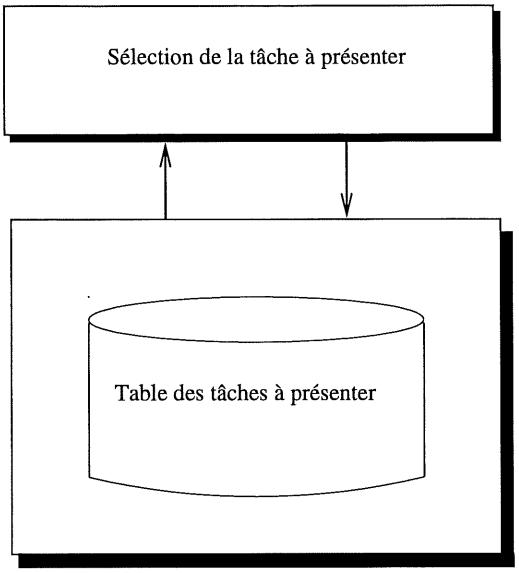

FIG. 24 – *Formulation de la tâche à présenter*

opérations d'analyse des interactions PPM-usager. Le processus *"anlyzeInteraction"* qui met en œuvre cette composante est développé selon le scénario suivant :

6) *anlyzeInteraction :* **Analyse de la tâche à présenter**

6.1) **Si** tâche déjà présentée (vérification dans la table des plans) **Alors**

6.1.1) **Si** cette tâche est présentée plus de trois fois (3) **Alors**

6.1.1.1) supprimer ancien plan de la table des PLANS
6.1.1.2) planifier cette tâche

6.1.2) **Si Non**

6.1.2.1) prendre le plan correspondant dans la table des PLANS **Fin Si**

6.2) **Si Non**
6.3) planifier cette tâche **Fin Si**

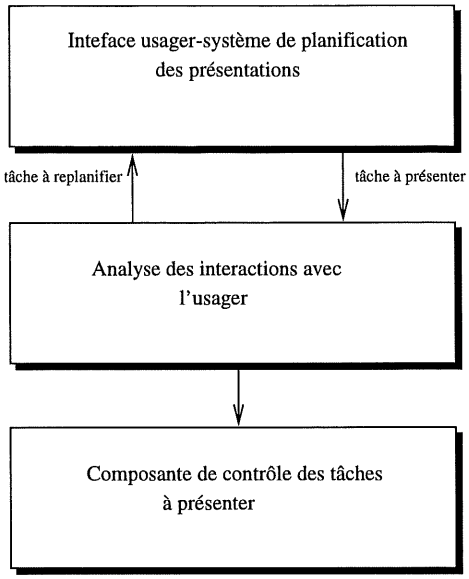

FIG. 25 – *Analyse de la tâche à présenter*

3.2.3 Composante Contrôle

La composante *Contrôle* communique la tâche à présenter et la description de l'état courant au planificateur et récupère de ce dernier le plan de la présentation générée (voir figure 26).

La composante *Contrôle* est très importante, d'une part, parce qu'elle se trouve au centre du système (*i.e.,* elle communique avec toutes les autres composantes), d'autre part, parce qu'il y a des extensions importantes qui peuvent venir se greffer sur cette composante. Par exemple, si on voulait offrir la possibilité à l'usager de formuler des tâches complexes composées de plusieurs sous-tâches, cette composante pourrait déterminer l'ordre de la planification des sous-tâches. Si on voulait doter PPM d'un personnage animé qui joue le rôle de présentateur, c'est encore avec cette composante que cet agent interagirait.

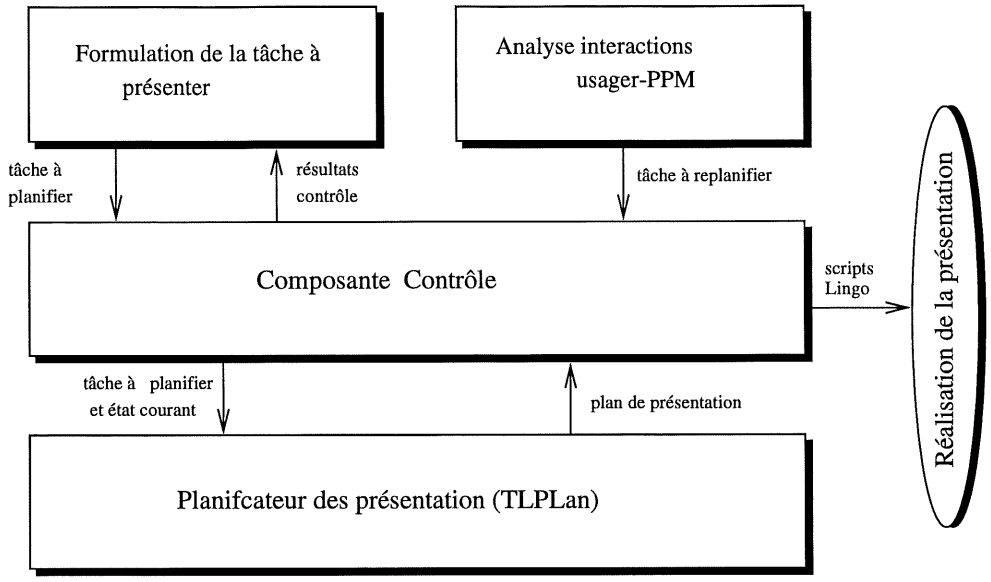

FIG. 26 – *Composante Contrôle*

La fonction pivot de cette composante est la communication avec *TLPlan*; cette communication se caractérise par l'envoi à *TLPLan* de l'état-but, l'état courant et la récupération d'un plan. Le scénario du processus*InteractionTLPlanPPM*, qui met en œuvre cette fonction, est le suivant :

7) *InteractionTLPlanPPM :* **Communication de la composante Contrôle avec** *TLPLan*

7.1) fichier destiné à *TLPLan* ← tâche à réaliser et état initial
7.2) exécution de *TLPLan*
7.3) plan ← lecture d'un fichier contenant le plan généré par *TLPLan*

La composante *Contrôle* communique le plan de la présentation générée par *TLPLan* à la composante Réalisation.

3.2.4 Composante Réalisation

Le rôle de la composante *Réalisation* est de composer un document multimédia à l'intention de l'usager à partir du plan de présentation et les médias stockés dans la

base de données. Le processus de réalisation se compose de deux tâches essentielles : la sélection des médias et la fusion et la coordination entre ces médias (voir figure 27).

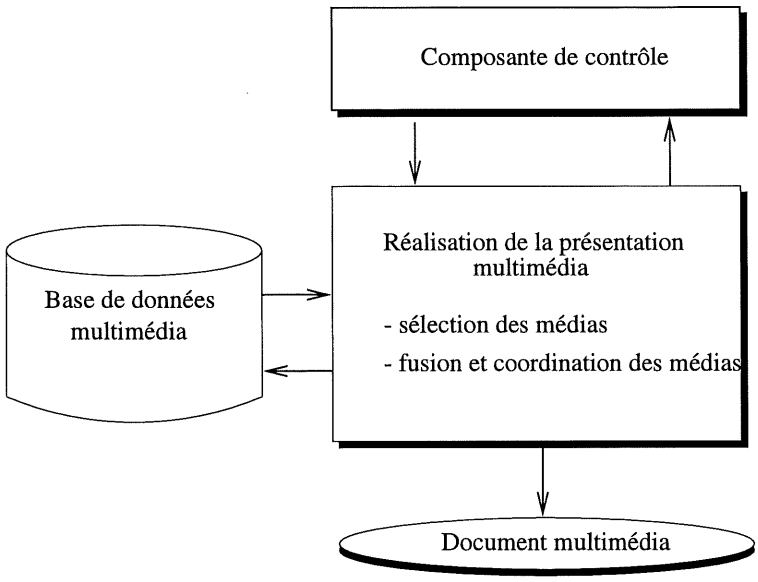

FIG. 27 – *Composante Réalisation*

Sélection des médias

À chaque action est associée un ensemble de médias. La composante *Réalisation* accède à ces médias dans la table *Actions*. Ces médias sont les casts qui vont composer le film *Director*. Dans un premier temps, ces casts sont importés de la base de données vers la table de distribution des casts, pour servir ensuite dans la composition de l'animation multimédia. Sachant que les médias occupent un espace mémoire non négligeable, les casts sont supprimés de la table de distribution des casts à la fin de chaque présentation. Le scénario suivant illustre la mise en œuvre du processus *importCast* :

8) *importCast :* **Importation des casts nécessaires pour composer le film**

8.1) **Tant que** la liste du plan n'est pas vide **répéter**

8.2) prendre l'action qui se trouve dans l'entête de la liste plan
8.3) accéder à la table ACTIONS.

8.3.1) casts vidéo ← média vidéo image de la table ACTIONS
8.3.2) casts son ← média son de la table ACTIONS
8.3.3) casts texte ← média texte de la table ACTIONS

8.2) **Fin Tant que**

Fusion et coordination des médias

À ce stade, PPM dispose de tous les médias nécessaires à la composition d'un film *Director*. La tâche suivante est la génération de la table de montage. Chaque action va être représentée par une trame, dans laquelle on ajoute les casts correspondants à cette action. Ainsi, la table de montage est construite trame par trame. Chaque trame représente une scène du film final et toutes les trames ensemble constituent la présentation multimédia attendue. Les casts prennent place dans des cellules pré-définies à l'avance et apparaîtront sur la scène selon les coordonnées spécifiées par les scripts *Lingo* qui leur sont associés. Le processus qui permet de mettre ensemble les médias dans une trame est appelé fusion des médias. Par exemple, l'action "retirer les piles" implique deux médias : une séquence vidéo et un fichier son. Ces médias sont ajoutés dans la même trame pour constituer une scène du film.

Des scripts *Lingo*, qui sont eux-mêmes des casts, coordonnent ces médias. Par exemple, la coordination entre la vidéo et le son impliqués dans l'action "retirer les piles de leur chambre", consiste à animer les deux casts ensemble et à ne passer à la trame suivante que lorsque l'animation du fichier de son est finie. Dans cet exemple, l'animation du cast de son dure plus longtemps que celle de la vidéo. En plus, on a intégré des contraintes temporelles à l'exécution des actions. Par exemple, l'action "soulever le couvercle de la

chambre des piles de l'appareil photo" dure au maximum 7 secondes. Au-delà de ce délai, PPM passe directement à la lecture de la trame suivante. Actuellement, ces délais sont spécifiés manuellement pour chaque action; il serait intéressant de les intégrer dans le planificateur.

De la même façon que dans un magnétoscope, la lecture séquentielle des trames fait apparaître une animation du film sur l'écran. Finalement, le film est monté d'une manière dynamique et en temps réel selon le plan de présentation générée. Nous avons utilisé les outils offerts par *Director* en matière de visualisation et de sonorisation des médias pour ne pas les réécrire. Le scénario de base du processus *scoreGeneration*, qui met en œuvre la fonction de fusion des médias, est :

9) *scoreGeneration :* **Génération de la table score**

9.1) **Pour chaque** action **répéter**
9.2) aller au frame suivant

9.2.1) cellule 1 ← cast vidéo
9.2.2) cellule 2 ← cast son
9.2.3) cellule 3 ← cast texte

Nous avons passé en revue tous les mécanismes de fonctionnement de PPM. Nous avons expliqué comment PPM génère automatiquement et en temps réel des présentations multimédias. Le processus de génération est basé sur la planification qui lui confère un caractère intelligent et lui permet de générer des présentations qui satisfont les besoins individuels des usagers. La planification permet à PPM d'extraire dans le volume des informations disponibles uniquement celles qui sont pertinentes aux présentations attendues. Pour montrer le bon fonctionnement de PPM, nous allons présenter dans le chapitre suivant deux exemples. Le premier consiste en la présentation d'un appareil photo. Le deuxième consiste en une interface de navigation d'un personnage animé dans un ensemble de chambres connexes.

Chapitre 4

Expérimentations

Ce chapitre se divise en deux parties. La première partie présente les deux exemples qui ont servi de bancs d'essais pour PPM. Ces deux exemples très différents montrent la polyvalence de l'idée de la planification des présentations multimédias et permettent de prendre conscience de l'étendue des applications possibles de cette idée. Dans le premier exemple, PPM doit générer des présentations multimédias qui apprennent à un usager l'utilisation d'un appareil photo. Dans le deuxième exemple, PPM doit générer une interface qui illustre la navigation d'un personnage de dessin animé dans un ensemble de chambres connexes.

4.1 Exemple 1 : Présentation d'un appareil photo

4.1.1 Description du problème

Dans cet exemple, il s'agit de montrer à un usager les manœuvres de base pour l'utilisation d'un appareil photo. Les présentations multimédias générées devraient apprendre à l'usager à prendre une photo et à exécuter toutes les actions nécessaires à la réalisation de cette manœuvre (*i.e.,* placer un film, retirer le film, placer les piles ou retirer les piles).

4.1.2 Modélisation du problème

Comme nous l'avons vu au chapitre 3, avant de créer une application sous PPM, il faut d'abord la modéliser. L'analyse de la description du problème nous permet de distinguer cinq manœuvres que l'usager peut opérer sur l'appareil photo : prendre une photo, retirer les piles de leur chambre, placer les piles dans leur chambre, retirer le film et placer le film. Si nous examinons ces manoeuvres, nous remarquons que, pour prendre une photo, il faut savoir enlever le cache de l'objectif et, bien entendu savoir le placer. Pour pouvoir exécuter les quatre autres manoeuvres, il faut savoir ouvrir le couvercle de la chambre des piles et celui de la chambre du film et savoir bien entendu les fermer.

Pour raffiner la modélisation, nous avons divisé en deux actions distinctes la manœuvre "ouvrir couvercle". Ces deux actions sont "ouvrir le couvercle" et "soulever le couvercle". Par conséquent, "fermer le couvercle" devient "baisser le couvercle" puis "fermer le couvercle". Ainsi, la liste complète des actions associées à l'utilisation d'un appareil photo sont : "ouvrir le couvercle", "soulever le couvercle", "baisser le couvercle", "fermer le couvercle", "enlever le cache de l'objectif", "placer le cache de l'objectif", "prendre une photo", "placer les piles", "retirer les piles", "placer le film" et "retirer le film".

Rappelons que les prédicats décrivent les actions de base, l'état du monde et la tâche à réaliser. Ces prédicats peuvent avoir pour arguments des variables qui sont instanciées lors de la planification. Par exemple, si nous considérons le couvercle de la chambre des piles, celui-ci peut être fermé ou ouvert. Donc, les propriétés "ouvert" et "fermé" décrivent l'état du couvercle. Cela est représenté dans le langage *TLPLan* par "`(Fermé x)`", et "`(Ouvert x)`" où x est une variable qui sera affectée durant la planification par "le couvercle de la chambre des piles" ou "le couvercle de la chambre du film". Il existe aussi des prédicats sans arguments tels que "`(caheObjEnlevé)`" pour indiquer que le cache de l'objectif est enlevé. La figure 28 présente la liste des prédicats tels que décrits

dans le langage utilisé par *TLPLan*. Nous remarquons que le nom du prédicat est suivi par le mot réservé "predicate" lui même suivi par le nombre d'arguments utilisés par ce prédicat. Les figures 29 et 30 présentent la liste des actions de base. Notons que le nom

```
(def-desribed-symbols
    '((Fermé predicate 1)
      (Soulevé predicate 1)
      (Baissé predicate 1)
      (Dedans predicate 1)
      (Vide predicate 1)
      (CacheOuvert predicate 0)
      (CachFermé predicate 0)))
```

FIG. 28 – *La liste des prédicats de la présentation d'un appareil photo*

de l'action est précédé du mot clé ":name" et que les listes des préconditions, additions et suppressions sont respectivement précédées des mots clés ":pre", ":add" et ":del".

4.1.3 Résultats

Selon le processus de génération des présentations multimédias, la première opération à effectuer consiste à créer la nouvelle application. Nous avons appelé cette application "Camera.V12". L'extension V12 fait référence au système de gestion des bases de données multimédias. Nous savons que les tables *Actions* et *Prédicats* reçoivent directement leurs données du fichier qui contient la modélisation de l'application. Par la suite, on spécifie les médias associés à chaque action. Ce n'est qu'à partir de ce moment que l'usager peut formuler les tâches à présenter. Par exemple, la formulation de la tâche "retirer les piles de leur chambre" consiste à donner la description formelle de l'état du système qui résulterait de la réalisation de cette tâche. Lorsqu'on retire les piles de leur chambre, le nouvel état se décrit par une chambre de piles vide. Cette situation peut être décrite formellement par le prédicat "(`Vide` chambrepiles)". Pour atteindre un état-but il faut partir d'un état initial. Pour chaque tâche à réaliser, on peut formuler un état initial

```
(add-strips-op
 :name '(Ouvrir ?x)
 :pre '((Fermé ?x))
 :add '((Baissé ?x))
 :del '((Fermé ?x)))

(add-strips-op
  :name '(Soulever ?x)
  :pre '((Baissé ?x))
  :add '((Soulevé ?x))
  :del '((Baissé ?x)))

(add-strips-op
 :name '(Baisser ?x)
 :pre '((Soulevé ?x))
 :add '((Baissé ?x))
 :del '((Soulevé ?x)))

(add-strips-op
  :name '(Fermer ?x)
  :pre '((Baissé ?x))
  :add '((Fermé ?x))
  :del '((Baissé ?x)))

(add-strips-op
  :name '(Placer ?x)
  :pre '((Vide ?x) (Souleve ?y))
  :add '((Dedans ?x))
  :del '((Vide ?x)))

(add-strips-op
  :name '(Retirer ?x)
  :pre '((Dedans ?x) (Soulevé ?y))
  :add '((Vide ?x))
  :del '((Dedans ?x)))
```

FIG. 29 – *La liste des actions de la présentation d'un appareil photo*

```
(add-strips-op
  :name '(OuvrirCache ?x)
  :pre '((CacheFermé))
  :add '((CacheOuvert))
  :del '((CacheFermé)))

(add-strips-op
 :name '(FermerCache ?x)
 :pre '((CacheOuvert))
 :add '((CacheFermé))
 :del '((CacheOuvert)))

(add-strips-op
 :name '(PrendrePhoto ?x)
 :pre '((CacheOuvert))
 :add '(())
 :del '(()))
```

FIG. 30 – *La liste des actions de la présentation d'un appareil photo (suite)*

particulier ou considérer que l'état courant du système est l'état initial, et ce, quelle que soit la tâche à réaliser.

Supposons que l'état initial est l'état courant pour réaliser la tâche "retirer les piles de leur chambre" et que cet état est décrit par "une chambre de piles vide". Dans ce cas, il est évident que le plan généré va d'abord expliquer comment placer les piles dans leur chambre et ensuite comment les retirer. Nous remarquons que la tâche "placer les piles dans leur chambre" n'a pas été explicitement spécifiée au système et qu'elle n'est pas nécessaire à l'explication de la tâche "retirer les piles de leur chambre". Nous concluons que dans ce type de présentation, l'état courant ne garantit pas la situation de départ idéale à l'explication des tâches formulées. Au contraire, les plans générés peuvent contenir des explications qui ne sont pas demandées par l'usager. Par conséquent, les présentations multimédias composées à partir de ces plans peuvent être vagues et imprécises. Pour livrer des explications précises, nous proposons dans cette application de spécifier un état initial particulier pour chaque tâche à réaliser. Par exemple, si l'état-intial est décrit par "les piles dans leur chambre" et "le couvercle de la chambre des piles est fermé" alors la présentation générée pour expliquer la tâche "retirer les piles

de leur chambre" contient uniquement des explications pertinentes, dont voici le plan de présentation :

```
(Ouvrir CouvercleChambreBatterie)
(Soulever CouvercleChambreBatterie)
(Retirer piles)
```

L'étape suivante consiste à importer les médias correspondants aux actions du plan dans la table des casts à partir de la table *Actions*, puis à générer la table du score. La lecture des trames générées permet à l'usager de suivre sur l'écran un film *Director* qui explique par le vidéo et le son la procédure pour retirer les piles de leur chambre. La figure 31 donne quelques extraits du film calculé suite à ce plan. Ces explications sont simples et réalistes et beaucoup plus agréables à suivre que de lire un papier avec des dessins pas toujours clairs et des références croisées souvent déroutantes. Cet exemple pourrait bien exister dans des kiosques de vendeurs d'appareils photo.

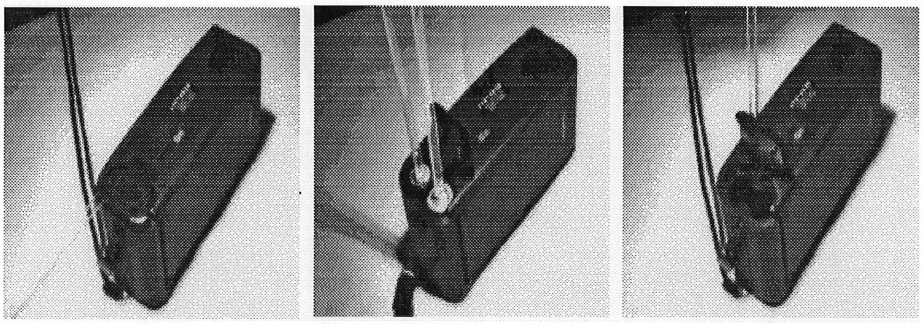

FIG. 31 – *Les films vidéo qui illustrent le retrait des piles*

4.2 Exemple 2 : Navigation d'un personnage animé

4.2.1 Description du problème

Dans cet exemple, il s'agit de fournir une interface graphique qui illustre la navigation d'un personnage de dessin animé (Toto) dans un ensemble de chambres connexes communiquant entre elles par des portes. Toto doit être capable de se rendre dans n'importe quelle autre chambre qu'on lui indique, à partir de la chambre où il se trouve. L'itinéraire de chaque déplacement n'est pas connu à l'avance. Au contraire, il est calculé par *TLPLan*.

4.2.2 Modélisation du problème

Ce problème est assez simple à modéliser. Selon la description du problème, nous identifions une première action qui consiste à passer d'une chambre à la chambre connexe. La présence des portes oblige Toto à savoir ouvrir et fermer une porte. Ces trois actions de base permettent à Toto de s'offrir n'importe quelle balade dans n'importe quel labyrinthe de chambres. La figure 32 présente la liste des prédicats qui décrivent les états de l'environnement de Toto.

```
(def-desribed-symbols
  '((Agent predicate 1)
   (PorteOuverte predicate 1)
   (PorteFermé predicate 1)
   (Connexes predicate 3)
   (dansChambre predicate 2)))
```

FIG. 32 – *La liste des prédicats de la présentation de la navigation du personnage animé Toto*

Le prédicat "(`agent` x)" sert à préciser de quel agent il s'agit, car il est possible que plusieurs agents partagent le même environnement. Les prédicats "(`porteOuvert` x)" et "(`porteFerme` x)" permettent de préciser l'état de la porte (fermée et ouverte). Le

prédicat "(connexe Porte chambre1 chambre2)" précise que la chambre " chambre1" est connexe à la chambre "chambre 2" et que les deux chambres communiquent par la porte "porte 1". La Figure 33 présente la liste des actions de base de la navigation de Toto.

```
(add-strips-op
 :name '(VaChambre ?ag ?ch1 ?ch2)
 :pre '((DansChambre ?ag ?ch1) (Connexes ?porte ?ch1 ?ch2) (porteOuverte ?porte))
 :add '((DansChambre ?ag ?ch2))
 :del '((DansChambre ?ag ?ch1)))

(add-strips-op
 :name '(OuvrirPorte ?ag ?porte ?chambre)
 :pre '((DansChambre ?ag ?chambre) (porteFermé ?porte))
 :add '((porteOuverte ?porte))
 :del '((porteFermé ?porte)))

(add-strips-op
 :name'(FermerPorte ?ag ?porte ?chambre)
 :pre '((DansChambre ?ag ?chambre) (porteOuverte ?porte))
 :add '((porteFermé ?porte))
 :del '((porteOuverte ?porte)))
```

FIG. 33 – *La liste des actions de la présentation de la navigation du personnage animé Toto*

4.2.3 Résultats

Contrairement au premier exemple, Toto part toujours de la chambre où il se trouve vers la chambre qu'on lui indique. Autrement dit, l'état initial pour Toto est toujours l'état courant du monde. Par conséquent, PPM doit garder une trace des transformations que subit l'environnement et doit posséder à n'importe quel moment la description de l'état actuel de l'environnement. Dans la table *Actions*, les trois actions correspondent à des films *Director*. Les actions ouvrir et fermer une porte sont des films qui montrent une porte s'ouvrir et se fermer accompagnés des sons simulant des claquements de portes. L'action de se déplacer dans la chambre connexe est un peu plus complexe car il y a quatre directions possibles pour se déplacer : nord, sud, est et ouest. Dans *Director*

ceci est traduit par quatre films chacun montrant Toto se déplacer dans l'une ou l'autre direction. À la suite de chaque déplacement de Toto, PPM restaure l'environnement de la nouvelle chambre à partir de la description actuelle de l'environnement. Chaque chambre peut avoir d'une à quatre portes (une porte par direction) et chaque porte peut être fermée ou ouverte (voir figure 34).

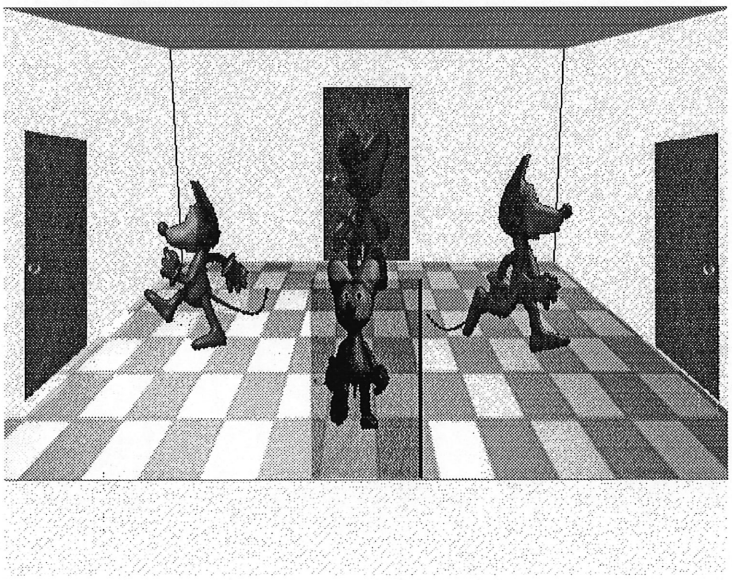

FIG. 34 – *L'environnement et les directions de déplacements du personnage animé Toto*

Supposons qu'au départ Toto est dans la chambre 11 (voir figure 11), alors l'état courant peut être décrit par un ensemble de conjonctions de prédicats dont voici un apperçu :

(`Agent` Toto)∧ (`dansChambre` Toto chambre11) ∧ (`Connexe` porte11 chambre11 chambre12)∧
(`Connexe` porte11 chambre12 chambre11)∧ (`Connexe` porte6 chambre11 chambre6)∧
(`Connexe` porte6 chambre6 chambre11)∧ (`Connexe` porte9 chambre11 chambre9)∧
(`Connexe` porte12 chambre11 chambre12)∧ (`Connexe` porte12 chambre12 chambre11)∧
(`Connexe` porte9 chambre9 chambre11)∧ (`porteFermé` porte11)∧ (`porteFermé` porte12)∧
(`porteFermé` porte6)...

Si l'usager désire que Toto se déplace dans la chambre 9, alors l'état but peut être décrit par le prédicat :

```
(dansChambre Toto chambre9)
```

Le plan généré par *TLPLan* est le suivant :

```
(ouvrirPorte porte11) (vaChambre Toto chambre11 chambre9)
```

L'environnement change après l'exécution du plan; Toto n'est plus dans la chambre 11, mais dans la chambre 9 et la porte 11 n'est plus fermée, mais ouverte. Comme nous l'avons expliqué dans le chapitre 2, *TLPLan* est doté d'un mécanisme efficace pour accélérer le processus de recherche d'un plan. Dans cet exemple, lors de la planification, l'exploration du chemin passant d'abord par la chambre 6 est une éventualité, sauf que dans *TLPLan*, on peut exprimer la stratégie de recherche suivante pour éviter des chemins inutiles :

```
'(always
  (forall (?porte) (?chX) (?chY) (contigues  ?porte  ?chX  ?chY)
    (implies (and (dansChambre  ?agent  ?chX)
                  (porteOuverte  ?porte)
                  (goal(dansChambre  ?agent  ?chY)))
             (next (dansChambre  ?agent  ?chY)))))
```

Cette règle écrite en logique temporelle spécifie que si l'état-but requiert d'aller dans une chambre connexe, alors il faut y aller tout de suite. Puisque dans cet exemple la chambre 9 est connexe à la chambre 11, cette règle force Toto d'aller directement dans la chambre spécifiée dans l'état-but. Lorsqu'on connaît la distance à parcourir pour se rendre dans une chambre connexe, une deuxième stratégie de contrôle permet à Toto d'aller dans la chambre la plus proche. Toto, étant dans la chambre 11, passe par la chmabre 9 plutôt que la chambre 6 pour se rendre dans la chambre 1. Une troisième stratégie de contrôle permet à Toto d'éviter les chambres impasses telles que la chambre 12, à moins qu'elles ne soient spécifiées dans les états-but. Ces stratégies simples et évidentes permettent dans certains cas d'éviter des recherches inutiles. Le lecteur intéressé trouvera plus de détails sur cette logique dans l'article [11].

Chapitre 5

Les travaux du DFKI

Dans ce chapitre nous proposons un résumé des travaux les plus récents et les proches du notre pour permettre au lecteur de situer notre travail dans ce domaine de recherche. Ces travaux sont réalisés par le DFKI (*centre allemand Saarbrücken de la recherche en intelligence artificielle*) qui, par l'originalité, des ses travaux est une équipe avant-gardiste dans ce domaine.

5.1 Le système WIP

Pour l'équipe du DFKI, l'aventure a commencé en 1989 par le développement d'un système de présentation multimédia intelligent appelé WIP (*Knowledge-Based Presentation of information*) [3]. De la même façon que dans PPM, WIP reçoit une tâche à présenter et génère une présentation multimédia. Les médias utilisés dans WIP sont le texte et le graphique, qui ne sont pas des médias enregistrés auparavant mais entièrement générés en temps réel pour satisfaire la tâche à présenter. Pour cela, WIP est doté d'un générateur de texte qui planifie chaque phrase à afficher et d'un générateur de graphique qui planifie chaque dessin nécessaire à la présentation. Ces deux générateurs sont sous contrôle du planificateur de la présentation qui se charge de coordonner le graphique et

le texte pour livrer une présentation cohérente [8].

5.2 Le système PPP

Le succès de la communication personne-personne est sûrement dû à la rhétorique et aux capacités didacticielles du présentateur. Cet aspect constitue l'âme du deuxième projet de l'équipe du DFKI. Il s'agit du présentateur PPP (*Personalized Plan-Based Presenter*) qui anime, montre, explique et commente verbalement et par le texte des tâches à présenter pour l'usager [6, 7, 9]. Dans la figure 35, PPP explique la mise en marche d'un modem. Grâce aux techniques du multi-fenêtrage, PPP se déplace autour de l'objet qu'il est entrain de présenter et à l'aide d'une règle, il pointe les endroits qu'il commente. Il est aussi doté de comportements destinés à représenter l'intention de l'usager. Par exemple, si PPM est en attente des données à présenter, au lieu de rester figé comme une image on le voit faire des pas de gymnastique et essayer d'être drôle. PPP peut afficher des signes de joie ou de fatigue et d'autres signes d'émotion.

5.3 Le système AIA

Avec l'*Internet*, sans le moindre doute le siècle de l'information est entrain de se préparer et même de se vivre. À travers tous les réseaux qui nous sont accessibles, nous sommes submergés par l'information et une des tâches importantes auxquelles nous sommes confrontés est la recherche de l'information. C'est pourquoi l'équipe du DFKI entame un projet ambitieux connu sous le nom AIA (*Adaptaive Infobahn Assistant*) [5]. AIA est un assistant des usagers qui naviguent à travers l'*Internet*. Il doit chercher, filtrer et présenter l'information dont les usagers ont besoin. En résumé, il leur évite le casse tête de la navigation dans l'océan d'information disponible. Ce système doit s'adapter

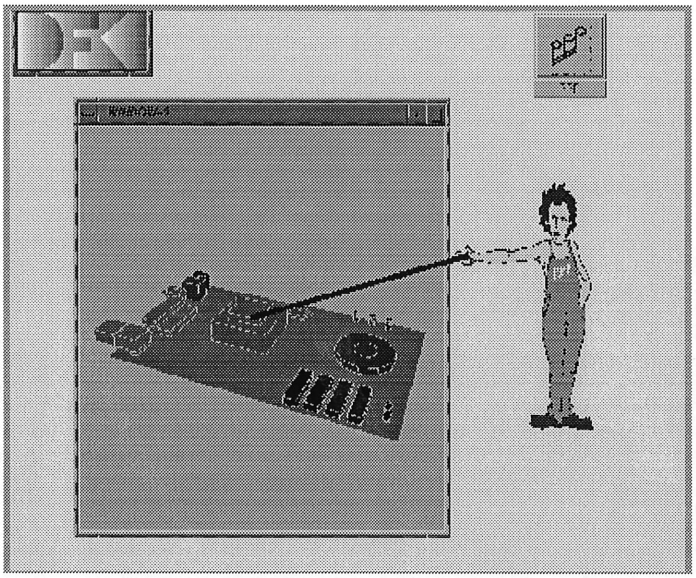

FIG. 35 – *PPP explique le fonctionnement d'un modem*

aux besoins particuliers de chaque usager dans un environnement complexe, dynamique et en perpétuelle évolution.

Par exemple, si un usager veut savoir quels sont les sites touristiques dans le Canada, AIA commence par trouver une carte du Canada puis procède à la recherche et la localisation des sites sur la carte. Après, il passe à la collecte de l'information touristique concernant chaque site. Enfin, il doit filtrer l'information selon le profil d'intérêt de l'usager pour lui communiquer l'information escomptée. AIA doit répondre aux besoins innombrables et imprévisibles des usagers. Dans ce même exemple, un utilisateur aimerait trouver les événements culturels organisés dans ces sites touristiques alors qu'un autre utilisateur peut demander de savoir les sites touristiques hôtes de salons informatiques.

Conclusion

Dans ce projet, les leçons apprises sont nombreuses et l'apport est plus que satisfaisant. En plus des techniques acquises telles que la planification et la programmation sous *Lingo*, ce travail nous a permis de réaliser le rôle prépondérant que peuvent jouer les techniques de l'intelligence artificielle dans le développement des systèmes informatiques futurs. Il n'est un secret pour personne que l'avenir de l'informatique est étroitement lié aux systèmes multimédias. Le traitement et la représentation de l'information de plus en plus complexe et dont le volume est sans cesse en croissance, nécessitent l'utilisation de nouvelles techniques. Il est évident que les techniques classiques de développement de systèmes où il faut tout prévoir à l'avance sont complètement inadaptées. La planification, les systèmes experts et les agents sont des alternatives à envisager. Tout au long de ce mémoire, nous avons essayé de montrer que la planification est une issue prometteuse.

Notre objectif consistait à développer un système capable de générer des présentations multimédias qui satisfont les besoins individuels des usagers. Par la réalisation de PPM, l'objectif est pleinement atteint. Le rôle de *TLPLan* a été largement exposé dans les chapitres 2 et 3. *TLPLan* est un logiciel complètement indépendant de PPM et son utilisation fait de PPM un système flexible et adaptable. Contrairement à un système classique où il faut coder les nouvelles présentations, dans PPM, l'usager n'a qu'à formuler ces nouvelles tâches et c'est *TLPLan* qui se charge de les planifier. PPM est adaptable,

puisque si on veut présenter une nouvelle application, on n'a qu'à fournir à PPM les actions de base et les médias correspondants.

Notre contribution se concrétise en deux points essentiels : l'utilisation d'un système auteur et l'utilisation d'un planificateur puissant. L'utilisation du système auteur nous a évité la réécriture de toutes les fonctions de manipulations des médias, et par conséquent elle nous facilite le développement de notre système. Sans *Director*, la réalisation de PPM aurait pu être une tâche très ardue. L'utilisation de *TLPLan* nous permet de générer des plans instantanément dès qu'une tâche est formulée. Nous savons que dans WIP et PPP, tout est planifié : le moindre pixel, le moindre mot est le résultat de la planification. Nous savons aussi que la planification est un processus très exigeant en temps. Alors nous avons toutes les raisons de croire que pour planifier des tâches complexes dans ces deux systèmes, il faut disposer de délais assez importants. Malheureusement, les très courtes démonstrations disponibles sur l'*Internet* ne nous permettent pas d'avoir des idées précises sur l'efficacité de ces deux systèmes. Bien entendu, le degré de raffinement dans WIP et PPP est beaucoup plus important que dans PPM, mais, sachant que le facteur temps est déterminant dans ce type de système, une question importante surgit : à quel degré de raffinement faut-il s'arrêter ? La réponse à cette question est sûrement liée au type de système que l'on veut réaliser. Cependant, nous savons avec certitude que PPM génère instantanément et en temps réel des présentations multimédias.

Certains aspects sont à améliorer dans notre système :

- la formulation des tâches : pour formuler une nouvelle tâche dans PPM, l'usager doit exprimer cette tâche par des conjonctions de prédicats, ce qui n'est pas très commode pour des usagers non familiers avec la logique;
- l'interaction PPM-usager : PPM ne permet aucune interaction avec l'usager au

moment de l'exécution de la présentation multimédia, sauf pour l'arrêter.

Pour remédier à ces deux faiblesses, nous proposons d'enrichir la composante de formulation de la tâche à présenter, de telle façon que l'usager puisse saisir du texte pour exprimer son besoin. Les tâches exprimées peuvent être des buts complexes. Dans ce cas, cette composante doit être capable de les décomposer en sous-buts plus simples et de déterminer l'ordre de leur planification.

Une deuxième amélioration consiste à offrir à l'usager la possibilité d'interagir avec PPM au moment de l'exécution de la présentation. Par exemple, dans le cas de la navigation de Toto, il serait intéressant que l'usager puisse agir sur l'environnement en ouvrant ou en fermant une porte, ou en plaçant des obstacles sur le chemin de Toto pour donner un caractère réaliste au système. Les environnements dans la vie réelle sont généralement instables, dynamiques et incertains. Dans ce cas, PPM doit être capable de percevoir le moindre changement dans son environnement et interagir avec un planificateur réactif capable de prendre les bonnes décisions dans n'importe quelle situation.

Les extensions que nous venons de proposer portent sur les aspects présentation (système multimédia) et planification. Pour enrichir le style de présentation, il faut sûrement se tourner vers les sciences sociales pour étudier le comportement de l'usager afin de déterminer ses intérêts et ses attirances. Finalement, les systèmes de présentations multimédias intelligents sont au début de leur aventure et le plus important des efforts est à venir.

Annexe A

Les programmes sources de PPM

Dans cette annexe, nous allons présenter quelques uns des scripts (*Lingo*) de la mise en œuvre de PPM. Ces scripts sont des extraits de programmes des processus de PPM présentés dans le chapitre 3. Nous commençons par présenter les scripts liés aux films, ensuite les scripts liés aux casts et enfin les scripts liés aux trames.

A.1 Les scripts liés aux films

```
global whichButton, quitClick, existingApp, newApp,choiceDone,
          NULL,cancelButton,doneButton, visible
on StartMovie

  openXlib "v12dbe"
  openXlib "v12Table"
  --
  -- Si present.v12 n'existe pas alors il faut exiter Builddb et stopMovie et
  -- remettre le commentaire tout de suite apres.
  --
  --builddb
  --disposeObjects
  --stopMovie
  set the visible of sprite 16 to FALSE
  set the visible of sprite  8  to FALSE
```

```
  set NULL = ""
  set whichButton = EMPTY
  set confirmOrCancelButton = EMPTY
  set appChoice = EMPTY
  set quitClick = 1
  set existingApp = 2
  set newApp = 3
  set doneButton = 4
  set cancelButton = 5
  set  visible = FALSE
  visibleOrNotConfirmMenu visible
end StartMovie

on deActivatesprites
  if  whichButton = newApp then
    set the castnum of sprite  5 to 8
  end if

  if  whichButton = existingApp then
    set the castNum of sprite 6 to 10
    set the visible of sprite  9 to FALSE
    set the visible of sprite 8  to FALSE
  end if
  updatestage
end

--
-- Ce handler s'occupe de la gestion des clicks de l'utilisateur a partir du menu
-- principal.
--
on processClick
  global whichButton, existingApp, newApp
  if whichButton = existingApp then
    disposeObjects
    play movie choiceDone
  end if

  if whichButton = newApp then
    set visible to FALSE
    visibleOrNotMainMenu visible
    findData        -- Cette fonction s'occupe de la creation de la base de la nouvelle presentation
```

```
  end if
end

--
-- quit the application
--
on disposeObjects
  -- Libération de l'espace alloué par les tables de la BD

  global db,  dbObj,  gAppt
  put objectP(gAppt)
  put objectP(gTable)
  put objectP(db)
  put objectP(dbObj)
  if objectP(gTable) then gTable(mDispose)
  if objectP(gAppt) then gAppt(mDispose)

  --stopMovie
end disposeObjects

on stopMovie
  --Adios
  if objectP(db) then
    db(mDispose)
  end if
  if objectP(dbObj) then
    dbObj(mDispose)
  end if
  closeXLib ("v12dbe")
  closeXLib  ("v12table")
  play done
end StopMovie

--
--  Build the data base
--
on BuildDb
  global db,presentName
  -- Creation de la Base de Données.
  if objectP(db) then mDispose(db)
  set db = v12dbe(mNew, the pathname & "present.v12","Create","")
```

```
  if NOT objectP(db) then
    if objectP(db) <> -12 then
      Alert("Can't create database: Presentation")
    else
      Alert("File already exist")
    end if
    --set the visible of sprite 16 to FALSE
    --set visible to TRUE
    --visibleOrNotMainMenu visible
    --deActivateSprites
    --nothingToDo
  end if

  ---- Table Applications.
  db(mCreateTable,"Applications" )
  db(mCreateField,"Applications","application","String",8)
  db(mCreateField,"Applications","Description","String",40)
  db(mCreateIndex,"Applications","appNDX","U","application","A")
  db(mBuild)
  db(mdispose)
end BuildDb

on importdb
  -- Import de Données
  -- Les donnees des differentes tables sont contenues dans des fichiers texte
  -- de la forme Champs <Tab> champs <TAB> ... champs <Return>

  global db,gTable,presentName, presentDesc
  if not objectP(db) then
    set db = V12dbe(mNew, the pathname & "present.v12","Readwrite","")
    if not  objectP(db) then
      Alert("Can't open database: Presentation")
      set the visible of sprite 16 to FALSE
      set visible to TRUE
      visibleOrNotMainMenu visible
      deActivateSprites
      nothingToDo
    end if
  end if

  -- Table des actions
```

```
  -- cette table  contient un champs Media chargement a l'aide de fileIo

  set gTable = v12table(mNew, db, "Applications")
  if NOT objectP(gTable) then
    Alert("Can't create table: Application")
    set the visible of sprite 16 to FALSE
    set visible to TRUE
    visibleOrNotMainMenu visible
    deActivateSprites
    nothingToDo
  end if
  --set presentName = "Camera"
  --set presentdesc =  "Presentation"
  put gtable(mAddRecord)
  put gtable(mSetfield, "application", presentName)
  put gtable(mSetfield, "description", presentDesc)
  put gtable(mUpdateRecord)
  gTable(mdispose)
  db(mdispose)

end importdb


--Initialisation de la base de donnees
on initdb
  -- Ouverture de la BD et intitialisation des tables
  global db, gAppT,n
  if not objectP(db) then
    set db = V12dbe(mNew, the pathname & "present.v12","Readwrite","")
    if not  objectP(db) then
      Alert("Can't open database: Presentation")
      set the visible of sprite 16 to FALSE
      set visible to TRUE
      visibleOrNotMainMenu visible
      deActivateSprites
      nothingToDo
    end if
  end if

  set gappT = v12table(mNew, db, "Applications")
  if NOT objectP(gappT) then
```

```
    Alert("Can't create table: Applications")
    exit
  end if
  gappT(msetindex, "appNDX")
  gappT(mselect)
  Put EMPTY into field "application"
  set n = gappT(mSelectCount)
  gappT(mGoFirst)

  repeat with i=1 to n
    put gappT(mgetfield, "Description") && RETURN after field "application"
    gappT(mGoNext)
  end repeat

end

--
-- Construire les fichiers:  actions, etats initiaux et buts a partir de la modelisation
-- se trouvant dans TLPLAN  " ?????world.lisp et ?????problems.lisp".
--
on findData
  global oldPath, gReadModel, visible,whichButton
  if objectP(gReadmodel) then   gReadModel(mDispose)
  set oldPath = "c:\tlplan\domains\camera\"
  put FileIO(mNew, "?read", oldPath & "*") into gReadModel
  if gReadModel = -43 then
    set visible to TRUE
    deActivatesprites
    visibleOrNotMainMenu visible
    set whichButton = EMPTY
    exit
  end if
  put greadmodel(mFileName)
  if gReadModel(mFileName) contains "world.lsp" then
    worldHandle
    if objectP(gReadmodel) then   gReadModel(mDispose)
    put FileIO(mNew, "?read", oldPath & "problems.lsp") into gReadModel
    problemsHandle
  else
    if gReadModel(mFileName) contains "problems.lsp" then
      problemsHandle
```

```
      if objectP(gReadmodel) then   gReadModel(mDispose)
      put FileIO(mNew, "?read", oldPath & "world.lsp") into gReadModel
      worldHandle
    else
      alert "Il faut lire un fichier de type ????world.lsp ou ?????problems.lsp"
      set visible to TRUE
      visibleOrNotMainMenu visible
      deActivateSprites
      nothingToDo
    end if
  end if
  set visible = TRUE
  visibleOrNotConfirmMenu
end findData

--
--  Rendre visible  ou invisible sprites pour confirmer ou abondonner la creation de
--  la base de donnees.
--

on visibleOrNotConfirmMenu visible

  put EMPTY into field "presentname"
  put EMPTY into field "presentdesc"
  if visible = TRUE then
    set the visible of sprite 10  to TRUE
    set the visible of sprite 11  to TRUE
    set the visible of sprite 12  to TRUE
    set the visible of sprite 13  to TRUE
    set the visible of sprite 14  to TRUE
    set the visible of sprite 15  to TRUE
  else
    set the visible of sprite 10  to FALSE
    set the visible of sprite 11  to FALSE
    set the visible of sprite 12  to FALSE
    set the visible of sprite 13  to FALSE
    set the visible of sprite 14  to FALSE
    set the visible of sprite 15  to FALSE
  end if
end visibleOrNotConfirmMenu
```

```
--
--  Rendre visible ou invisible les sprites pour confirmer ou abondonner la creation de
--  la base de donnees.
--

on visibleOrNotMainMenu visible
  if visible = TRUE then
    set the visible of sprite 3  to TRUE
    set the visible of sprite 4  to TRUE
    set the visible of sprite 5  to TRUE
    set the visible of sprite 6  to TRUE
    set the visible of sprite 7  to TRUE
    --set the visible of sprite 8  to TRUE
    --set the visible of sprite 9  to TRUE
  else
    set the visible of sprite 3  to FALSE
    set the visible of sprite 4  to FALSE
    set the visible of sprite 5  to FALSE
    set the visible of sprite 6  to FALSE
    set the visible of sprite 7  to FALSE
    set the visible of sprite 8  to FALSE
    set the visible of sprite 9  to FALSE
  end if
end visibleOrNotMainMenu

--
--  Abondonner ou confirmer la creation de la base de donnees.
--
on confirmOrCancel
  global confirmOrCancelButton, presentation, presentName, presentDesc,doneButton ,cancelButton

  if confirmOrCancelButton = doneButton then
    set presentName = word 1 of line 1 of field "presentname"
    if   presentName = NULL then
      nothing
    end if

    set presentDesc = line 1 of  field "presentdesc"
    set visible to FALSE
    visibleOrNotconfirmMenu visible
    set the visible of sprite 16 to TRUE
```

```
    importdb
    buildDataBase
    importDataBase
    set the visible of sprite 16 to FALSE
    set visible to TRUE
    visibleOrNotMainMenu visible
    deActivateSprites      -- Remettre le boutton new App non enfonce
  end if

  if confirmOrCancelButton = cancelButton then
    alert("Creation d'application Abondonée")
    set visible to FALSE
    visibleOrNotconfirmMenu visible
    set visible to TRUE
    visibleOrNotMainMenu visible
    set the visible of sprite 5 to 8      -- Remettre le boutton new App non enfonce
    nothingToDo
  end if
end confirmOrCancel

--
-- Creer la base contenant les tables: ACTIONS, INITIAL STATES ET GOALS a partir du
-- Fichier ??????world.lisp
--
on worldHandle
  global  lineContent, gReadModel, charRead,actionsList, EndPredicate,predicateList
  set predicateList = []
  set actionsList  = []
  set lineContent = NULL
  set contain = TRUE
  repeat while contain
    put gReadModel(mReadLine) into lineContent
    if lineContent contains "def-described-symbols" then set contain = FALSE

    if lineContent = "" then
      alert " Fi-chier vide ou ne contenant pas de predicats..???????"
      set visible to TRUE
      visibleOrNotMainMenu visible
      deActivateSprites
      nothingToDo
    end if
```

```
  end repeat

  put lineContent

  set charRead = NULL
  repeat while charRead<> charToNum("'") -- code ascii 39
    put gReadModel(mReadChar) into charRead

    if charRead = -1 then
      alert "Fichier vide ou ne contenant pas de predicats  ???????"
      set visible to TRUE
      visibleOrNotMainMenu visible
      deActivateSprites
      nothingToDo
    end if

  end repeat
  put gReadModel(mReadChar) into charRead
  if charRead = -1 then
    alert "Fichier vide ou ne contenant pas de predicats  ???????"
    set visible to TRUE
    visibleOrNotMainMenu visible
    deActivateSprites
    nothingToDo
  end if
  if charRead <> charToNum("(") then
    alert "la syntaxe de ce fichier n'est pas correcte  ???????"
    set visible to TRUE
    visibleOrNotMainMenu visible
    deActivateSprites
    nothingToDo
  end if
  set EndPredicate = TRUE
  repeat while EndPredicate
    ProcessPredicateLine
  end repeat
  processActionLine
end worldHandle

--
-- EXtraire les predicats du fichier ?????world.lsp
```

```
--

on  ProcessPredicateLine
  global charRead, predicateList,endPredicate
  set predicate  = EMPTY
  set charRead = EMPTY
  put gReadModel(mReadChar) into charRead
  if charRead = -1 then
    alert "Fichier vide ou ne contenant pas de predicats???????"
    set visible to TRUE
    visibleOrNotMainMenu visible
    deActivateSprites
    nothingToDo
  end if
  repeat while charRead = charToNum(" ") or charRead =charToNum("(")
    put gReadModel(mReadChar) into charRead
    if charRead = -1 then
      alert "Fichier vide ou ne contenant pas de predicats???????"
      set visible to TRUE
      visibleOrNotMainMenu visible
      deActivateSprites
      nothingo
    end if
    --stop
  end repeat
  set predicate = predicate & numToChar(charRead)
  put gReadModel(mReadChar) into charRead
  if charRead = -1 then
    alert "Fichier vide ou ne contenant pas de predicats???????"
    stopMovie
  end if
  repeat while charRead <> charToNum(" ")

    set predicate = predicate & numToChar(charRead)
    put gReadModel(mReadChar) into charRead
  end repeat
  add(predicateList, predicate)
  repeat while (charRead < 48 or charRead >57)
    put gReadModel(mReadChar) into charRead
    if charRead = -1 then
      alert "Fichier vide ou ne contenant pas de predicats???????"
```

```
      stopMovie
    end if
  end repeat
  set nombreParam = nombreParam & numToChar(charRead)
  put gReadModel(mReadChar) into charRead
  repeat while (charRead >= 48 and charRead <=57)
    set nombreParam = nombreParam & numToChar(charRead)
  end repeat
  add(predicateList, nombreParam)

  repeat while charRead <> charToNum(")")
    if numToChar(charRead) = ")" then set parenthese = parenthese + 1
    put gReadModel(mReadChar) into charRead
    if charRead = -1 then
      alert "Fichier vide ou ne contenant pas de predicats???????"
      stopMovie
    end if
  end repeat
  if parenthese = 3 then set EndPredicate = FALSE
  put predicateList
end ProcessPredicateLine

--
-- Chercher les actions
--
on ProcessactionLine
  global  lineContent, gReadModel, charRead, actionsList
  set actionsList  = []
  set lineContent = NULL
  set EndFile = TRUE
  repeat while EndFile
    put gReadModel(mReadLine) into lineContent
    if lineContent = "" then set endFile = FALSE
    if lineContent contains "add-strips-op" then  findAction

    if lineContent = "" and actionsList =[]  then
      alert "fin fichier ou ne contenant pas d'actions..???????"
      stopMovie
    else
      if lineContent = ""  then
        gReadModel(mDispose)
```

```
        --nothingToDo
      end if
    end if

  end repeat

  put lineContent
end processActionLine

on findAction
  set charRead = NULL
  repeat while charRead<> charToNum("'") -- code ascii 39
    put gReadModel(mReadChar) into charRead
    if charRead = -1 then
      alert "Fichier vide ou ne contenat pas d'actions???????"
      nothingToDo
    end if
  end repeat

  put gReadModel(mReadChar) into charRead
  repeat while charRead <> charToNum(")")
    set action = action & numToChar(charRead)
    put gReadModel(mReadChar) into charRead
  end repeat
  set action = action & numToChar(charRead)
  put add(actionsList, action)
end findAction

--
--
--
on problemsHandle
  global  wordContent, gReadModel, charRead,goalList ,stateList
  set goalList = []
  set stateList  = []
  set wordContent = NULL
  set  wordContent = "|"
  repeat while wordContent <> ""
    put gReadModel(mReadWord) into wordContent
    if wordContent contains "setf" then findGoalState
    if wordContent = ""  and stateList = [] then
```

```
      alert "Fin fichier ou ne contenant pas des etats init ou des buts..???????"
      nothingToDo
    end if
  end repeat
  nothingToDo
end problemsHandle

on findGoalState
  global  wordContent, gReadModel, charRead,goalList ,stateList
  --trouver state
  set initState = NULL
  set goal = NULL
  set state = FALSE
  put gReadModel(mReadWord) into wordContent

  if wordContent contains "state" then
    set state = TRUE
  else
    if wordContent contains "goal" then
      set state = FALSE
    else
      alert "fichier ?????world.lsp incorrect"
      nothingToDo
    end if
  end if

  set charRead = NULL
  repeat while charRead<> charToNum("'") -- code ascii 39
    put gReadModel(mReadChar) into charRead
    if charRead = -1 then
      alert "Fichier vide ou ne contenat pas de state ou de goal???????"
      nothingToDo
    end if
  end repeat

  put gReadModel(mReadChar) into charRead
  if charRead = -1 then
    alert "Fichier vide ou ne contenat pas de state ou de goal???????"
    nothingToDo
  end if
  if charRead <> charToNum("(") then
```

```
    alert "Fichier ???????world.lsp incorrect"
    nothingToDo
  end if
  skipSpaces
  set parentheses = 0
  repeat while parentheses <= 1
    --put gReadModel(mReadChar) into charRead
    repeat while charRead <> charToNum(")")
      if charRead = charToNum("(") then
        set parentheses = 0
      end if
      if state = TRUE then
        set initState = initState & numToChar(charRead)
      else
        set goal = goal & numToChar(charRead)
      end if

      put gReadModel(mReadChar) into charRead
    end repeat
    --set initState = NULL
    --set goal= NULL
    set parentheses = parentheses + 1
    if parentheses <= 1 then
      if state = TRUE then
        set initState = initState & numToChar(charRead)
      else
        set goal = goal & numToChar(charRead)
      end if
    end if

    skipSpaces
  end repeat
  if state = TRUE then
    put add(stateList, initState)
  else
    put add(goalList, goal)
  end if

end findGoalState

--
```

```
--
--
on skipSpaces
  global charRead
  put gReadModel(mReadChar) into charRead
  repeat while charRead = charToNum(" ")
    put gReadModel(mReadChar) into charRead
  end repeat
end skipSpaces

on nothingToDo
  nothing
end nothingToDo


--
--build data base
--
on BuildDatabase
  global dbObj,presentName
  -- Creation de la Base de Donnée
  set presentName ="camera"
  set fichier = presentName & ".v12"
  if objectP(dbobj) then mdispose(dbObj)
  set dbObj = v12dbe(mNew, the pathname & fichier,"Create","")
  if Not objectP(dbObj) then
    if objectP(dbObj) <> -12 then
      Alert("Can't create database")
    else
      alert("DataBase already exist")
    end if

    set the visible of sprite 16 to FALSE
    set visible to TRUE
    visibleOrNotMainMenu visible
    deActivateSprites
    nothingToDo

  end if
```

```
---- Table Actions
put dbObj(mCreateTable,"Actions" )
put dbObj(mCreateField,"Actions","action","String",20)
put dbObj(mCreateField,"Actions","Description","String",100)
put dbObj(mCreateField,"Actions","sound","String",20)
put dbObj(mCreateField,"Actions","Picture","String",20)
put dbObj(mCreateField,"Actions","Texte","String",20)
put dbObj(mCreateField,"Actions","Status","String",1)
put dbObj(mCreateIndex,"Actions","actionNDX","U","Action","A")

---- Table predicats
put dbObj(mCreateTable,"Predicate" )
put dbObj(mCreateField,"Predicate","predicate","String",20)
put dbObj(mCreateField,"Predicate","Param","string",2)
put dbObj(mCreateField,"Predicate","Status","String",1)
put dbObj(mCreateIndex,"Predicate","predNDX","U","predicate","A")

---- Table des buts
put dbObj(mCreateTable,"Goals" )
put dbObj(mCreateField,"Goals","goal","string",200)
put dbObj(mCreateField,"Goals","Description","String",200)
put dbObj(mCreateField,"Goals","Status","String",1)
put dbObj(mCreateIndex,"Goals","goalNDX","U","Goal","A")

---- Table des etats initiaux
put dbObj(mCreateTable,"InitialSt" )
put dbObj(mCreateField,"InitialSt","state","string",200)
put dbObj(mCreateField,"InitialSt","Description","String",200)
put dbObj(mCreateField,"InitialSt","Status","String",1)
put dbObj(mCreateIndex,"InitialSt","stateNDX","U","state","A")

---- Table des plans
put dbObj(mCreateTable,"Plans" )
put dbObj(mCreateField,"Plans","State","string",200)
put dbObj(mCreateField,"Plans","Goal","string",200)
put dbObj(mCreateField,"Plans","plan","string",255)
put dbObj(mCreateField,"Plans","Description","String",255)
put dbObj(mCreateField,"Plans","Status","string",1)
put dbObj(mCreateIndex,"Plans","goalPNDX","D","Goal","A")

---- Table du plan de la presentation
```

```
  put dbObj(mCreateTable,"PlanPres" )
  put dbObj(mCreateField,"PlanPres","Plan","string",255)
  put dbObj(mCreateField,"PlanPres","Description","String",255)
  put dbObj(mCreateField,"PlanPres","Status","String",1)
  put dbObj(mCreateIndex,"PlanPres","planNDX","U","plan","A")
  dbobj(mdispose)
end


-- Import de Données
on importDatabase

  -- Les donnees des differentes tables sont contenues dans des fichiers texte
  -- de la forme Champs <Tab> champs <TAB> ... champs <Return>

  global dboj,gtable, actionsList, goalList , predicateList, NULL,presentName
  set presentName ="camera"
  set fichier = presentName & ".v12"
  if not objectP(dbObj) then
    set dbObj = V12dbe(mNew, the pathname & "Camera.v12","Readwrite","")
    if NOT objectP(dbObj) then
      Alert("Can't open database: Camera")
      set the visible of sprite 16 to FALSE
      set visible to TRUE
      visibleOrNotMainMenu visible
      deActivateSprites
      nothingToDo
    end if
  end if

  ---- Table des actions
  -- cette table  contient un champs Media chargement a l'aide de fileIo

  set gTable = v12table(mNew, dbObj, "Actions")
  if NOT objectP(gTable) then
    Alert("Can't create table: Actions")
    set the visible of sprite 16 to FALSE
    set visible to TRUE
    visibleOrNotMainMenu visible
    deActivateSprites
    nothingToDo
```

```
end if
put actionsList && RETURN after  field "kad"
set fLen = count(actionsList)

-- Lecture des tuples
repeat with  i=1 to fLen  -- exit only if end of file
  set Action = getAt(actionsList, i)
  set  Description = NULL
  set  Sound = NULL
  set  Picture = NULL
  set  Texte = NULL
  set Status = "A"

  -- Ajout d'un tuple
  put gtable(mAddRecord)

  put gtable(mSetfield, "Action", Action)
  put gtable(mSetfield, "Description", Description)
  put gtable(mSetfield, "Sound", Sound)
  put gtable(mSetfield, "Picture", Picture)
  put gtable(mSetfield, "Texte", Texte)
  put gtable(mSetfield, "Status", Status)
  put gtable(mUpdateRecord)
  put action && RETURN after field "kad"
end repeat

--gTable(mDispose)

-- Lecture des tuples

gtable(msetindex, "actionNDX")
gtable(mselect)
Put EMPTY into field "list1"
put "ACTIONS" && RETURN after field "list1"
set n = gtable(mSelectCount)
gtable(mGoFirst)

repeat with i = 1 to n

  put  gtable(mgetfield, "action") &&
  gtable(mgetfield, "Description") &&
```

```
  gtable(mgetfield, "Sound") &&
  gtable(mgetfield, "Picture") &&
  gtable(mgetfield, "Texte") &&
  RETURN after field "list1"


  gtable(mGoNext)
end repeat

gTable(mdispose)

---- Table goals
set gTable = v12table(mNew, dbObj, "Goals")
if NOT objectP(gTable) then
  Alert("Can't create table: Goals")
  set the visible of sprite 16 to FALSE
  set visible to TRUE
  visibleOrNotMainMenu visible
  deActivateSprites
  nothingToDo
end if

set fLen = count(goalList)

-- Lecture des tuples
repeat with i=1 to fLen  -- exit only if end of file
  set goal = getAt(goalList, i)
  set  Description = NULL
  set Status = "A"

  -- Ajout d'un tuple
  put gtable(mAddRecord)

  put gtable(mSetfield, "Goal", goal)
  put gtable(mSetfield, "Description", Description)
  put gtable(mSetfield, "Status", Status)
  put gtable(mUpdateRecord)
end repeat

--gTable(mDispose)
```

```
Put EMPTY into field "list2"
put "Goals" && RETURN after field "list2"
set n = count(goalList)
Repeat with i = 1 to n

  put  gtable(mgetfield, "Goal") &&
  gtable(mgetfield, "Description") &&
  gtable(mgetfield, "status") &&
  RETURN after field "list2"
  gtable(mGoNext)

end repeat

gtable(mdispose)



---- Table états initiaux
set gTable = v12table(mNew, dbObj, "InitialSt")
if NOT objectP(gTable) then
  Alert("Can't create table: Initial States")
  set the visible of sprite 16 to FALSE
  set visible to TRUE
  visibleOrNotMainMenu visible
  deActivateSprites
  nothingToDo
end if

set fLen = count(goalList)

-- Lecture des tuples
repeat with i=1 to fLen  -- exit only if end of file
  set state = getAt(stateList, i)
  set  Description = NULL
  set Status = "A"

  -- Ajout d'un tuple
  put gtable(mAddRecord)

  put gtable(mSetfield, "State", state)
  put gtable(mSetfield, "Description", Description)
```

```
    put gtable(mSetfield, "Status", Status)
    put gtable(mUpdateRecord)
  end repeat

  --gTable(mDispose)

  gtable(msetindex, "stateNDX")
  gtable(mselect)
  Put EMPTY into field "list3"
  put "Initial States" && RETURN after field "list3"
  set n = gtable(mSelectCount)
  gtable(mGoFirst)
  repeat with i = 1 to n

    put  gtable(mgetfield, "State") &&
    gtable(mgetfield, "Description") &&
gtable(mgetfield, "status") &&
    RETURN after field "list3"

    gtable(mGoNext)
  end repeat

  gtable(mdispose)


  ---- Table predicats
  set gTable = v12table(mNew, dbObj, "Predicate")
  if NOT objectP(gTable) then
    Alert("Can't create table: Predicate")
    set the visible of sprite 16 to FALSE
    set visible to TRUE
    visibleOrNotMainMenu visible
    deActivateSprites
    nothingToDo
  end if

  set fLen = count(predicateList)

  -- Lecture des tuples
  repeat with i=1 to fLen  -- exit only if end of file
    set predicate = getAt(predicateList, i)
```

```
    set i =i +1
    set param = getAt(predicateList, i)
    set  Description = NULL
    set Status = "A"

    -- Ajout d'un tuple
    put gtable(mAddRecord)


    put gtable(mSetfield, "Predicate", predicate)
    put gtable(mSetfield, "param", param)
    put gtable(mSetfield, "Status", Status)
    put gtable(mUpdateRecord)
  end repeat

  --gTable(mDispose)

  gtable(msetindex, "predNDX")
  gtable(mselect)
  Put EMPTY into field "list4"
  put "predicate" && RETURN after field "list4"
  set n = gtable(mSelectCount)
  gtable(mGoFirst)
  repeat with i = 1 to n

    put  gtable(mgetfield, "predicate") &&
    gtable(mgetfield, "Param") &&
    gtable(mgetfield, "status") &&
    RETURN after field "list4"

    gtable(mGoNext)
  end repeat

  gtable(mdispose)
  dbObj(mdispose)
end importdatabase
```

les programmes de khal.dir

```
global  whichButton, presentChoice, goalChoice, upDateChoice, automaticChoice, interactifChoice,
contentChoice,whatChoice,actionsList, markerList,planIs,specifChoice
 ,predicateChoice, actionChoice, planspresented, SoundTime, SoundFile

on StartMovie

  -- Ouverture des librairies pour la gestion de la base de données
  openXlib "v12dbe"
  openXlib  "v12table"

  initdb -- Script à appeler pour ouvrir la base de données et les différents handler des tables
  init
end -- start Movie


on init
  set the visible of sprite 22 to FALSE
  set the visible of sprite 15 to FALSE
  set the visible of sprite 23 to FALSE
  set the visible of sprite 21 to FALSE
  set the visible of sprite 19 to FALSE
  set the visible of sprite 20 to FALSE
  set the visible of sprite 18 to FALSE
  set the visible of sprite 16 to FALSE
  set whichButton = EMPTY
  set presentChoice = 1
  set goalChoice = 2
  set updateChoice = 3
  set automaticChoice = 4
  set interactifChoice = 5
  set whatChoice = 6
  set contentChoice = 7
  set specifChoice = 8
  set actionChoice = 9
  set predicateChoice = 10
  set plansPresented = []

  -- liste des actions de la modelisation defenie dans TLPlan.

  set markerList = ["M1","M3",
                                "M4","M5","M6","M7","M8",
```

```
                                "M9","M10","M11",
                                "M12","M13","M14","M15","M16"]
  set soundFile =["closebat.wav", "downbat.wav", "inbat.wav", "openbat.wav", "outbat.wav","upbat.wav"]
  set soundTime = [7,6,7,7,6,6]
  -- Liste de tous les Arguments
  Put EMPTY into field "update"

  put "Graphical Updates" && RETURN &&
  "Content Updates" && RETURN  after field "update"

  Put EMPTY into field "content"

  put "Actions Updates" && RETURN &&
 "Predicate Updates" && RETURN &&
 "Initial states updates" && RETURN &&
 "Goal Updates" && RETURN &&
  "Plans Updates" && RETURN after field "content"
  specification
end

on specification
  put Empty into field "goal"
  ggoalsT(msetIndex,"goalNDX")
  ggoalsT(mselect)

  set n = ggoalsT(mSelectCount)
  ggoalsT(mGoFirst)
  Repeat with i = 1 to n

    put  ggoalsT(mgetfield, "Description") &&
    RETURN after field "goal"
    ggoalsT(mGoNext)

  end repeat

end

-- on Stop Movie

on disposeObjects
  -- Libération de l'espace alloué par les handlers de la BD
```

```
  gactionsT(mdispose)
  ggoalsT(mdispose)
  ginitialstT(mdispose)
  gplansT(mdispose)
  gPlanpresT(mdispose)
  dbObj(mdispose)
  closexlib "v12dbe"
  closexlib "v12table"
end disposeObjects

on stopMovie

  play done

end stopMovie

--Initialisation de la base de donnees
on initdb
  -- Ouverture de la BD et intitialisation des tables
  global dbObj, gactionsT, ggoalsT, ginitialstT, gplansT, gplanpresT
  if objectP(dbObj) then dbObj(mdispose)
  set dbObj = V12dbe(mNew, the pathname & "Camera.v12","Readwrite","")
  if NOT objectP(dbObj) then
    Alert("Can't open database")
    exit
  end if


  set gactionsT = v12table(mNew, dbObj, "Actions")
  if NOT objectP(gactionsT) then
    Alert("Can't create table: Actions")
    exit
  end if

  set ggoalsT = v12table(mNew, dbObj, "Goals")
  if NOT objectP(ggoalsT) then
    Alert("Can't create table: Goals")
    exit
  end if

  set ginitialstT = v12table(mNew, dbObj, "InitialSt")
```

```
  if NOT objectP(ginitialStT) then
    Alert("Can't create table: Initial State")
    exit
  end if

  set gplansT = v12table(mNew, dbObj, "Plans")
  if NOT objectP(gplansT) then
    Alert("Can't create table: Plans")
    exit
  end if

  set gplanpresT = v12table(mNew, dbObj, "Planpres")
  if NOT objectP(gplanPresT) then
    Alert("Can't create table: Plan Presentation")
    exit
  end if
end

-- activation des sprites corrspondant aux clicks effectues

on activateSprites

  if whichButton = automaticChoice then
    set the puppet of sprite 4 to TRUE
    set the castNum of sprite 4 to 29
    updateStage
    -- the rest is coming
  end if


  if whichButton = interactifChoice then
    set the puppet of sprite 7 to true
    set the castNum of sprite 7 to 31
    updateStage
    -- the rest is coming
  end if

  if whichButton = presentChoice then
    set the puppet of sprite 10 to true
    set the castNum of sprite 10 to 35
    updateStage
```

```
    set the visible of sprite 22 to TRUE
    -- the rest is coming
  end if

  if whichButton = whatChoice then
    set the puppet of sprite 13 to true
    set the castNum of sprite 13 to 33
    updateStage
    set the visible of sprite 21 to TRUE
    -- the rest is coming
  end if
end activateSprites

-- des-activation des sprites corrspondant aux clicks effectues

on deActivateSprites

  if whichButton = automaticChoice then
    set the puppet of sprite 4 to true
    set the castNum of sprite 4 to 28
    updateStage
    set the visible of sprite 15  to FALSE
    -- the rest is coming
  end if

  if whichButton = interactifChoice then
    set the puppet of sprite 7 to true
    set the castNum of sprite 7 to 30
    updateStage
    set the visible of sprite 15  to FALSE
    -- the rest is coming
  end if

  if whichButton = presentChoice  or whichButton = contentChoice
      or whichButton = upDateChoice  then
    set the puppet of sprite 10 to true
    set the castNum of sprite 10 to 34
    updateStage
    set the visible of sprite 15 to FALSE
    set the visible of sprite 22 to FALSE
    set the visible of sprite 23 to FALSE
```

```
    -- the rest is coming
  end if


  if whichButton = whatChoice  or whichButton = goalChoice  then
    set the puppet of sprite 13 to true
    set the castNum of sprite 13 to 32
    updateStage
    set the visible of sprite 15 to FALSE
    set the visible of sprite 21 to FALSE
    -- the rest is coming
  end if
end activateSprites

--
--
--
on ProcessClick
  global ggoal, gplan,gapp, planIs
  if whichButton = goalChoice then
    gplansT(msetindex, "goalNDX")
    set state = "(batterin) (Coverbatup)"
    gplansT(msetcriteria, "goal", "=", ggoal)
    gplansT(mselect)
    put gplansT(mgetfield, "goal")
    put gplansT(mgetfield, "state")
    put gplansT(mgetfield, "plan")

    if gplansT(mselect) <> 0 or gplansT(mgetfield, "plan") = -1223 then
      Alert("Plan non existant dans la BD, le systeme va lui generer un plan")
      set state = "(batteryin) (coverbatClosed)"
      writeFile state, ggoal
      open "lisp.exe"
      set gReadObject = -43
      repeat while gReadObject=-43
        startTimer

        repeat while the timer < 60*5
          nothing
        end repeat
        put fileIo(mNew, "read", "c:\allegro\plan.txt") into gReadObject
```

```
      end repeat
      if objectP(gReadObject) then gReadObject(mDispose)
      readFile

      addNewPlan state, ggoal
      runplan
      --exit
    else
      set gplan =gplansT(mgetfield, "plan")

      put gplan
      constructPlan
      runPLan
      --exit
    end if

  end if
  if whichButton = contentChoice then
    disposeObjects
    if gapp = "actions"  then play frame "actions" of movie "khal.dir"
    if gapp = "Predicate"  then play frame "Predicates" of movie "khal.dir"
    if gapp = "Plans"  then play frame "plans" of movie "khal.dir"
    if gapp = "Goal"  then play frame "goals" of movie "khal.dir"
    if gapp = "initial"  then play frame "initialsStates" of movie "khal.dir"
  end if

end

--
--
--

on constructPlan
  global gplan
  set planIs =[]
  repeat with i = 1 to the number of words in gplan
    set   plan = word i of gplan
    add(planIs,plan)
  end repeat
  put planIs
```

```
end

--
-- Handler for reading the plan commuicated with TLPLan
--
on readFile
  global gReadObject, planIs
  set planIs =  []
  -- Select the file for reading
  put fileIo(mNew, "read", "c:\allegro\plan.txt") into gReadObject
  put gReadObject
  --methode   the contents
  put gReadObject(mReadChar) into  text
  put text
  repeat while text >= 0 and text < 200 then
    if text = 13 or text = 31 or text = 10  or text = 0 then
      nothing
    else
      set action = EMPTY
      if  numToChar(text) = "(" then
        repeat while numToChar(text) <> ")"
          set action = action & numToChar(text)
          put gReadObject(mReadchar) into  text
        end repeat
        set action = action & numToChar(text)
        add(planIs, action)
      end if
    end if
    put gReadObject(mReadchar) into  text
  end repeat
  --if objectP(gReadObject) then gReadObject(mDispose)
  put planis
  gReadObject(mDelete)
  if the result < 0 then alert "Plan.txt ne peut etre supprime"
end readFile
--
-- Handler for writing the goal and the initial state to commuicat with TLPLan
--
on writeFile state, ggoal
  global gWriteObject,planIs
  set planIs =  []
```

```
  -- Select the file for Writing
  put fileIo(mNew, "write", "c:\allegro\gands.txt") into gWriteObject
  if objectP(gWriteObject) then gWriteObject(mDispose)
  put fileIo(mNew, "append", "c:\allegro\gands.txt") into gWriteObject
  gWriteObject(mWriteString, "("& state&")" &RETURN)
  gWriteObject(mWriteString, "("&ggoal&")")
  if objectP(gWriteObject) then gWriteObject(mDispose)
end WriteFile
--
--
--
on runPlan
  global planIs,listAvi,ListCastAvi,listBmp, listCastBmp, listWav, listCastWav, ListPlansPresented
  CommonSubGoal
  set the visible of sprite 15 to FALSE
  set the visible of sprite 13 to FALSE
  set listAvi =[]
  set listCastAvi =[]
  set listBmp=[]
  set listCastBmp =[]
  set listWav =[]
  set listCastBmp =[]
  set n = count(planIs)
  set avi =35
  set bmp =45
  set wav = 55
  gactionsT(msetindex, "actionNDX")

  repeat with i= 1 to n
    --
    --  Prendre les images correspondantes a l'action
    --
    gactionsT(msetcriteria, "action", "=", getAt(planIs,i))
    gactionsT(mSelect)
    set k = avi +i
    if gactionsT(mgetField,"picture") <> -1 then
      put gactionsT(mgetField,"picture")
      if  getPos(listavi, gactionsT(mgetField,"picture" ))  <> 0 Then
        set j = avi + getPos(listavi, gactionsT(mgetField,"picture"))
        addAt(listCastAvi,i,j)
        put  listCastAvi
```

```
    else
      addAt(listCastAvi,i,k)
      put listCastAvi
    end if
    put addAt(listAvi,i,gactionsT(mgetField,"picture"))
    put listavi
    importFileInto cast k ,"C:\director\learning\"& gactionsT(mgetField,"picture")
  end if
  --
  --  Prendre les sons correspondantes
  --
  set k = Bmp + i
  put gactionsT(mgetField,"text")
  if gactionsT(mgetField,"text") <> -1 then
    if  getPos(listBmp, gactionsT(mgetField,"text" ))  <> 0 Then
      set j = avi + getPos(listBmp, gactionsT(mgetField,"text"))
      addAt(listCastBmp,i,j)
    else
      addAt(listCastBmp,i,k)
    end if
    addAt(listBmp,i,gactionsT(mgetField,"text"))
    importFileInto cast k ,"C:\director\learning\"& gactionsT(mgetField,"text")
  end if
  --
  --  Prendre les sons correspondantes sons
  --

  put gactionsT(mgetField,"Sound")
  if gactionsT(mgetField,"Sound") <> -1 then
    addAt(listwav,i,gactionsT(mgetField,"Sound"))
    put listwAV
  end if
end repeat

repeat with i=1 to n
  go to frame "M1"

  if listcastavi <> [] then
    if getAt(listcastavi,i) <>  ""  then
      set the castnum of sprite 2 to getAt(listcastavi,i)
    end if
```

```
    end if

    if  listcastBmp <> [] then
      if  getAt(listcastBmp,i) <> "" then
        set the castnum of sprite 3 to getAt(listcastBmp,i)
      end if
    end if

    updateStage
    put getAt(listWav,I)
    global soundFile,SoundTime
    set temp = 0
    set temp = getAT(soundTime,getPos(SoundFile, getAt(listWav,I)))
    if getAt(listWav,I) <> -1   then  sound playFile 1, "C:\director\learning\"& GetAt(listWav,i)
    if getAt(listcastavi,i) contains ".avi" then
      puppetSprite 2, TRUE
      set the movieTime of sprite 2 to 0
      set the movieRate of sprite 2 to 1
    end if
    if getAt(listWav,I) <> ""   then  sound playFile 1, "C:\director\learning\"& GetAt(listWav,i)
    startTimer
    repeat while the timer < 60*temp
      nothing
    end repeat
  end repeat
  ---------------------------nouveau



  repeat with i=36 to 59
    erase cast i
  end repeat

  set the visible of sprite 13 to TRUE
  set the visible of sprite 15 to TRUE
  go to frame "iniT"
end

--
--
--
```

```
on addNewPlan state,ggoal
  global planIs, plansPresented
  set n = count(planIs)
  set plan = EMPTY
  repeat with i=1 to n
    if plan = empty then
      set plan = plan & getAt(planIs, i)
    else
      set plan = plan && getAt(planIs, i)
    end if

  end repeat
  gplansT(mSetIndex, "goalPNDX")
  gplansT(mAddRecord)
  gplansT(msetField, "plan",plan)
  gplansT(msetField, "goal",ggoal)
  gplansT(msetField, "state",state)
  gplansT(mUpdateRecord)
  --set planIs =[]
end

--
--
--
on commonSubGoal
  global planIs, plansPresented, myPlan
  set plan = EMPTY
  set n = count(planis)
  repeat with i=1 to n
    if plan = EMPTY then
      set plan = plan & getAt(planIs, i)
    else
      set plan = plan && getAt(planIs, i)
    end if

  end repeat
  if plansPresented = [] then
    add(plansPresented, plan)
    exit
  end if
  set n = count(plansPresented)
```

```
  set subgoal = EMPTY
  repeat with i =1 to n
    if getAt(plansPresented, i) contains plan then
      alert(" Ceci est un sous but d'un but deja presenté")
      exit
      set subgoal = getAt(plansPresented, i)
      set i =n + 1
    end if
  end repeat
end

--if subgoal = EMPTY then
-- add(plansPresented, plan)
-- exit
--end if
--set the visible of sprite 20 to TRUE
---set the visible of sprite 18 to TRUE
--set the visible of sprite 16 to TRUE
--updateStage
--set myPlan = plan--
--pause
--end

--on suite
--global plan, planIs, myPlan, confirmation
--if confirmation = "oui" then
--set the itemDelimiter = " "
--put the number of items in myplan into n
--repeat with i =1 to n
--put item i of myplan into action
--set k = getPos(planIs, action)
--deleteAt(planIs, k)
--end repeat
--end if
--set confirmation = EMPTY
--end
```

A.2 Les scripts liés aux casts

```
on mousedown
  set the castNum of sprite 14 to 6
  updateStage
  repeat while the mouseDown
  end repeat
end

on mouseUp
  if the frameLabel = "Goals"  or the frameLabel = "initialStates" then
    global goalUpdateChoice, addGoal, addPredicate, predicateList
    set the castNum of sprite 14 to 5
    updateStage
    if goalUpdateChoice = addGoal then
      set the castNum of sprite 2 to 62
      set the castNum of sprite 3 to 65
      set the castNum of sprite 12 to 12
      updateStage
      put EMPTY into field "GoalsPred"
      set predicateList = []
      initPredicate
      set goalUpdateChoice = addPredicate
      exit
    end if

    if goalUpdateChoice = addPredicate then
      global gnomPredicate, gnbpredicate,predicateList, gpredicateLine
      if  getPos(predicateList, gnomPredicate) then exit
      set gpredicateLine = gpredicateLine + 1
      put gnbpredicate && gnompredicate   && RETURN after field  "goalspred"
      hilite Line gnbpredicate of Field "goalspred"
      add(predicateList,gnomPredicate)
    end if
  end if
  --
  -- cas de plans
  --
  if the frameLabel = "Plans"  then
    global planUpdateChoice, addPlan, addAction, actionList
    set the castNum of sprite 14 to 5
```

```
    updateStage
    if planUpdateChoice = addplan then
      set the castNum of sprite 2 to 24
      set the castNum of sprite 3 to 27
     -- set the castNum of sprite 12 to 12
      updateStage
      put EMPTY into field "plansAction"
      set actionList = []
      initAction "actionlist"
      set planUpdateChoice = addAction
      exit
    end if

    if planUpdateChoice = addAction then
      global gnomAction, gnbAction,actionList, gactionLine
      if  getPos(actionList, gnomAction) then exit
      set gActionLine = gActionLine + 1
      put gnbAction && gnomAction   && RETURN after field  "plansAction"
      hilite Line gnbAction of Field "plansAction"
      add(actionList,gnomAction)
    end if
  end if
end
on mousedown
  set the castNum of sprite 12 to 15
  updateStage
  repeat while the mouseDown
  end repeat
end

on mouseUp
  set the castNum of sprite 12 to 12
  updateStage
  if the frameLabel = "Goals" or the frameLabel = "initialStates" then
    global  goalUpdateChoice, addPredicate, addgoal,predicateList, gpredicateLine
    if   goalUpdateChoice = addPredicate then
      put EMPTY into  field  "goalspred"
      set predicateList = []
      set gpredicateLine = 0
    end if
    exit
```

```
  end if
  --
  -- cas concernant des plans
  --
  if the frameLabel = "Plans" then
    global  planUpdateChoice, addAction, addplan,actionList, gactionLine
    if   planUpdateChoice = addAction then
      put EMPTY into  field  "plansaction"
      set actionList = []
      set gactionLine = 0
    end if
    exit
  end if
end

on mouseDown
  global gnomPredicate, gnbpredicate,predicateList, gpredicateLine,goalUpDateChoice,
              addPredicate, addGoal
  set the castNum of sprite 12 to 15
  set the castNum of sprite 2 to 62
  set the castNum of sprite 3 to 65
  updateStage
  repeat while the mouseDown
  end repeat
end

on mouseUp
  global gnomPredicate, gnbpredicate,predicateList, gpredicateLine,goalUpDateChoice,
              addPredicate, addGoal, modifyGoal
  set the castNum of sprite 12 to 13
  updateStage
  if goalUpdateChoice = addGoal then
    set gpredicateLine = count(predicateList)
    set goalUpdateChoice = addPredicate
    initPredicate
    set modifyGoal = TRUE -- s'il s'agit d'une modification.
    exit
  end if
  --put EMPTY into field "GoalsPred"
  --set predicateList = []
  set gpredicateLine = gpredicateLine + 1
```

```
  put gnbpredicate && gnompredicate   && RETURN after field  "goalspred"
  hilite Line gnbpredicate of Field "goalspred"
  add(predicateList,gnomPredicate)
end

on mousedown
  set the castNum of sprite 15 to 8
  updateStage
  repeat while the mouseDown
  end repeat
end

on mouseUp
  set the castNum of sprite 15 to 7
  updateStage
  if the frameLabel = "Goals" or the frameLabel = "initialStates" then
    global goalUpDateChoice, addpredicate, addGoal, gpredicateLine
         , predicatList, modifyGoal
    if goalUpdateChoice = addPredicate then

      set the castNum of sprite 2 to 60
      set the castNum of sprite 3 to 63
      set the castNum of sprite 12 to 13
      updateStage
      put EMPTY into field "GoalsPred"
      set predicateList = []
      set gPredicateLine = 0
    end if
  end if
  --
  -- Cas d'un plan
  --
  if the frameLabel = "Plans" then
    global planUpDateChoice, addAction, addGoal, gactionLine
         , actionList
    if planUpdateChoice = addaction then

      set the castNum of sprite 2 to 22
      set the castNum of sprite 3 to 25
      --set the castNum of sprite 12 to 13
      updateStage
```

```
      put EMPTY into field "plansaction"
      set actionList = []
      set gActionLine = 0
    end if
  end if
  init
  whichFieldInit
end
```

A.3 Les scripts liés aux Trames

```
on exitFrame
  pause
end
on initFieldActsList
  global gActionsT, dbObj
  put EMPTY into field "ActsList"
  set gactionsT =   v12table(mNew, dbObj, "Actions")
  if NOT objectP(gactionsT) then
    Alert("can't Create Table : Actions")
    exit
  end if

  gactionsT(msetindex, "actionNDX")
  gactionsT(mselect)
  set n = gactionsT(mSelectcount)
  repeat with i = 1 to n
    put gactionsT(mgetfield, "action") &&
 RETURN after field "ActsList"
    gactionsT(mGoNext)
  end repeat
end

on exitFrame
  pause
end

on exitFrame
  go to the frame
```

```
end
on mouseUp
  continue
end
```

Bibliographie

[1] E. André. Intellimedia: Making Multimedia Usable by Exploiting AI Methods. *ACM Journal Computing Surveys*, 27(4):560–563, 1995.

[2] E. André. Presentation: General Models and Issues. In I. F. Cruz, J. Marks, and K. Wittenburg, editors, *Proceedings of the Workshop on Effective Abstractions in Multimedia Layout, Presentations, and Interaction in conjunction with ACM Multimedia '95.* 1995. Also available as Electronic Proceedings via http://www.cs.tufts.edu/~isabel/mmwsproc.html.

[3] E. Andre, W. Finker, W. Grag, T. Rist, A. Schauder, and W. Wahsler. WIP: The Automatic Synthesis of Multimodal Presentation. In M. Maybury, editor, *Intelligent multimedia interfaces*, pages 75–93. MIT Press, 1993.

[4] E. André, W. Graf, J. Heinsohn, B. Nebel, H.-J. Profitlich, T. Rist, and W. Wahlster. PPP - Personalized Plan-Based Presenter. Document D-93-5, DFKI, Saarbrücken, 1993.

[5] E. André, W. Graf, J. Müller, H.-J. Profitlich, T. Rist, and W. Wahlster. AiA: Adaptive Communication Assistant for Effective Infobahn Access. Document, DFKI, Saarbrücken, 1996.

[6] E. André, J. Müller, and T. Rist. The PPP Persona: A Multipurpose Animated Presentation Agent. In *Advanced Visual Interfaces*, pages 245–247. ACM Press, 1996.

[7] E. André, J. Müller, and T. Rist. WIP/PPP: Automatic Generation of Personalized Multimedia Presentations. In *ACM Multimedia 96*, pages 407–408. ACM Press, November 1996.

[8] E. Andre and T. Rist. Generating Coherent Presentation Employing Textual and Visual Material. *AI Review, Special volume of natural languageand vision processing*, 9(2-3):147–165, 1995.

[9] E. Andre, W., and T. Rist. Coping with Temporal Constraints in Multimedia Presentation Planning. In *Thirteen National Conference AAAI*, volume 1, pages 142–147, Portland, Oregon, 1996.

[10] F. Bacchus and F. Kabanza. Tlplan (version 2.0) user's manual. Document, tutorial et manuel d'utilisation, Université de Sherbrooke et Université de Waterloo, 1995.

[11] F. Bacchus and F. Kabanza. Using Temporal Logic to Control Search in a Forward Chaining Planner. In *In Proc. 3rd European Workshop on Planning (EWSP)*, 1995.

[12] M. Bordegoni, G. Faconti, M.T. Maybury, T. Rist, S. Ruggieri, P. Trahanias, and M. Wilson. A Standard Reference Model for Intelligent Multimedia Presentation Systems. *The International Journal on the Development and Application of Standards for Computers, Data Communications and Interfaces*, 1997.

[13] J.D Burger and R.J. Marshall. The Application of Natural Models to Intelligent Multimedia. In M. Maybury, editor, *Intelligent Multimedia interfaces*, pages 174–196. MIT Press, 1993.

[14] S.K. Feiner and R. McKeown. Automating the Generation of Coordinated Multimedia Explanation. In M. Maybury, editor, *Intelligent Multimedia interfaces*, pages 117–138. MIT Press, 1991.

[15] S. Francklin and A. Graesser. Is it an Agent or just a Program?: A taxonomy for Autonomous Agent. In *Third International Workshop on Agent Theories, Archiecture, and Languages*, pages 21–35. Spring-Verlag, 1997.

[16] G.J Holzmann. *Design and Validation of Computer Protocols.* AAAI Press, 1991.

[17] MACROMEDIA inc. *Director Academic V 4.0.* Prentice-Hall, 1995.

[18] C. Karagiannidis, A. Koumpis, and C. Stephanidis. Deciding what, when, why and how to adapt in intelligent multimedia presentation systems. In G.P. Faconti and T. Rist, editors, *Proceedings of ECAI'96 Workshop Towards a Standard Reference Model for Intelligent Multimedia Presentation Systems.* ECAI'96 - J.von Neumann Computer Society, Bathori u. 16, H-1054 Budapest, August 1996.

[19] V. Lifschistz. On the Semantics of Strips. In Morgan Kauffman, editor, *Readings in planning*, pages 523–530. AAAI Press, 1991.

[20] J.D. Mackinlay. Automating Design of Graphical Presentation of Rational Information. *ACM Transactionon graphics*, 5(2):110–114, 1986.

[21] M.T. Maybury. *Intelligent Multimedia Interfaces.* AAAI Press, 1993.

[22] Integration multimedia. V12 DataBase Engine. document, User manual Version 1.1, Integration new media Inc,, 1995.

[23] T. Rist and E. André. Designing Coherent Multimedia Presentations. In G. Salvendy and M.J. Smith, editors, *Human-Computer Interaction: Software and Hardware Interfaces (Proc. of HCI-93)*, volume 19B, pages 434–439. Elsevier, Amsterdam, London, 1993.

[24] T. Rist, E. André, and J. Müller. Adding Animated Presentation Agents to the Interface. In *Proceedings of the 1997 International Conference on Intelligent User Interfaces*, pages 79–86, Orlando, Florida, 1997.

[25] S.F Roth and J. Mattis. Automatic Graphics Presentation. In *Proceeding of IEEE Conference on AI Applications*, pages 90–97, Miami Beach, FL, 1991.

[26] D.E Wilkins. Practical Planning- Extending the classical AI planning paradigm. In Morgan-Kofmann, editor, *Readings in planing.* AAAI Press, 1991.